Unicenter

TCPaccess Communications Server
RPC/XDR Programmer Reference

Version 6.0

Computer Associates

The Software That Manages eBusiness

Booklist

Contents

Chapter 1: Introduction to RPC/XDR

Chapter 2: Using Remote Procedure Calls

Chapter 3: XDR: Technical Notes

Chapter 4: Using rpcgen

Appendix A: RPC Manual Pages

Appendix B: XDR Manual Pages

Appendix C: RPC Library Header Files

Appendix D: RPC Log

Appendix E: Sample JCL

Appendix F: Sample RPC Programs

Index

Chapter 1

Introduction to RPC/XDR

This chapter introduces and defines the Unicenter TCPaccess Communications Server RPC/XDR packages. It introduces the RPC/XDR packages and defines the terms Remote Procedure Call (RPC) and External Data Representation (XDR).

The Remote Procedure Call (RPC) package defines a procedure calling model for distributed applications. The External Data Representation (XDR) defines a standard representation for data in the network to support heterogeneous network computing.

Unicenter TCPaccess RPC/XDR lets you create custom distributed applications and network services using the mainframe and the resources of the network. Both client and RPC functionality exist in this implementation. This means an application on the mainframe using RPC/XDR can not only provide resources to the network, but can access resources and initiate activity on the network as well. RPC/XDR includes both RPC and XDR library routines.

The C language interface to the RPC/XDR library is compatible with the UNIX operating system reference standard, which facilitates development of network services on the mainframe. You can select from the RPC package a TCP or UDP transport on which to run your application.

Remote Procedure Call (RPC)

RPC is an independent set of functions used for accessing remote nodes on a network. Using RPC network services, applications can be created in much the same way a programmer writes software for a single computer using local procedure calls. The RPC protocols extend the concept of local procedure calls across the network. This means that you can develop distributed applications for transparent execution across a network.

External Data Representation (XDR)

XDR is a vendor-independent method of representing data. By using the XDR standard data representation convention, systems do not have to understand and translate every data format that may exist on the network; there is only the one convention. Data is translated into XDR format before it is sent over the network and, at the reception point, is translated into the data convention used there. This means that you can integrate new computer architectures into the network without requiring the updating of translation routines.

The new architecture simply includes a routine that translates its data format into XDR format and the new member of the network is ready to go. Using XDR, data can be accessed or exchanged among machines of various hardware and software architectures without any translation or interpretation problems. Word lengths, byte ordering, and floating-point representations appear to be the same to all nodes in the network.

Chapter 2

Using Remote Procedure Calls

This chapter describes the use of Remote Procedure Calls (RPCs). It includes these sections:

- Higher Layers of RPC—Describes the highest and intermediate layers of RPC
- Lowest Layer of RPC—Describes the lowest level of RPC programs
- Useful RPC Features—Discusses some use of select on the server side, broadcast RPC, batching and authentication
- Programming Examples—Provides examples of the use of version numbers, a Unix remote file copy program, and callback procedures

This guide assumes a working knowledge of network theory. It is intended for programmers who wish to write network applications using Remote Procedure Calls (RPC), and who want to understand the RPC mechanisms usually hidden by the rpcgen protocol compiler rpcgen is described in detail in Using rpcgen.

Note: Before attempting to write a network application, or to convert an existing non-network application to run over the network, you may want to understand the material in this chapter. However, for most applications, you can bypass the material presented here by using rpcgen. The section Generating XDR Routines contains the complete source for a working RPC service—a remote directory listing service that uses rpcgen to generate XDR routines as well as client and server stubs.

Remote Procedure Calls (RPCs) are high-level communications mechanisms. RPC presumes the existence of low-level networking mechanisms (such as TCP/IP and UDP/IP), and implements on them a logical client-to-server communications system designed specifically for the support of network applications.

With RPC, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and returns the procedure call to the client.

RPC Layers

The RPC interface can be seen as being divided into three layers.

The Highest Layer

The highest layer is totally transparent to the operating system, machine, and network on which it is run. Think of this level as a way of using RPC, rather than as a part of RPC itself.

Programmers who write RPC routines usually make this layer available to others by way of a simple C language front end that entirely hides the networking.

At this level, a program can simply make a call to rnusers(), a C routine that returns the number of users on a remote machine. Users are not explicitly aware of using RPC—they simply call a procedure, just as they would call malloc().

The Middle Layer

The middle layer is really the heart of RPC. Here, the user does not need to consider details about sockets, the UNIX system, or other low-level implementation mechanisms. They simply make remote procedure calls to routines on other machines. The inherent value of this layer is its simplicity. It allows RPC to pass the "hello world" test.

The middle layer routines are used for most applications. RPC calls are made with the system routines registerrpc(), callrpc(), and svc_run().

The first two of these are the most fundamental: registerrpc() obtains a unique system-wide procedure identification number, and callrpc() actually executes a remote procedure call. At the middle layer, a call to rnusers() is implemented by way of these two routines.

The middle layer, however, is rarely used in serious programming due to its simplicity. It does not allow timeout specifications or the choice of transport. It allows no UNIX process control or flexibility in case of errors. It does not support multiple types of call authentication. You rarely need all these types of control, but one or two of them is often necessary.

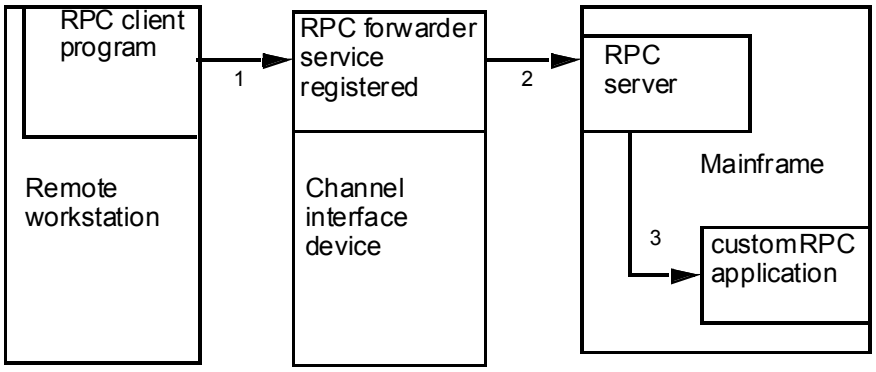
The Lowest Layer

The lowest layer lets you control these details, and for that reason it is often necessary. Programs written at this level are also most efficient, but this is usually not an issue, since RPC clients and servers rarely generate heavy network loads.

Note: Although this document discusses only the interface to C, remote procedure calls can be made from any language. Even though this document discusses how RPC is used to communicate between processes on different machines, RPC works just as well for communication between different processes on the same machine.

The RPC Paradigm

The following is a diagram of the RPC paradigm:



1. RPC client program issues an RPC call to a remote procedure.
2. RPC forwarder recognizes call to a mainframe service and forwards RPC call over the channel.
3. Mainframe RPC server accepts the call and forwards the request to the RPC application.

Higher Layers of RPC

This section describes the highest and middle layers of RPC.

Highest Layer

Suppose you are writing a program that needs to know how many users are logged into a remote machine. You can do this by calling the RPC library routine rnusers(), as illustrated here:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char **argv;
{
    int num;

    if (argc != 2)
    {
        fprintf(stderr, "usage: rnusers hostname\n");
        exit(1);
    }
    if ((num = rnusers(argv[1])) < 0)
    {
        fprintf(stderr, "error: rnusers\n");
        exit(-1);
    }
    printf("%d users on %s\n", num, argv[1]);
    exit(0);
}
```

RPC Service Library

On a UNIX system, RPC library routines such as rnusers() are in the RPC services library librpcsvc.a. Therefore, the previous program should be compiled on UNIX using this format:

```
example% cc program.c -lrpcsvc
```

Note: Unicenter TCPaccess RPC/XDR does not provide this RPC services library. However, applications may be written on MVS, which accesses these functions on other machines.

Some of the available RPC service library routines are listed in the following table:

Routine	Description
rnusers	Returns number of users on remote machine.
rusers	Returns information about users on remote machine.
have disk	Determines if remote machine has disk.
rstats	Gets performance data from remote kernel.
rwall	Writes to specified remote machines.
yppasswd	Updates user password in Yellow Pages.

Other RPC services, such as ether(), mount(), and spray(), are not available to the C programmer as library routines. They do, however, have RPC program numbers (which are discussed in the next section), so they can be invoked with callrpc(). Most of the other RPC services also have compilable rpcgen(1) protocol description files.

Note: The rpcgen protocol compiler radically simplifies the process of developing network applications. See the chapter "Using rpcgen" for detailed information about rpcgen and the rpcgen protocol description file. rpcgen is not currently provided with RPC/XDR.

Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions callrpc() and registerrpc(). Using this method, the number of remote users can be found by using this program. It can be used with the RPC/XDR code supplied if the #defines for the ruser program are used.

```
#include <stdio.h>
#include <rpc.h>
/* #include <rusers.h> /* not included with RPC/XDR */

#define RUSERSPROG 10002
#define RUSERSVERS 2
#define RUSERPROC-NUM 1

main(argc, argv)
    int argc; char **argv;
{
    unsigned long nusers;
    int stat;

    if (argc != 2)
    {
        fprintf(stderr, "usage: (char) nusers hostname\n");
        exit(-1);
    }

    if (stat = callrpc(argv[1], RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
                       xdr_void, 0, xdr_u_long, &nusers) != 0)
       {
           clnt_perrno(stat);
           exit(1);
       }
    printf("%d users on %s\n", nusers, argv[1]);
    exit(0);
}
```

Unique RPC Procedure Definition

Each RPC procedure is uniquely defined by a program number, version number, and procedure number.

- The program number specifies a group of related remote procedures, each of which has a different procedure number
- Each program also has a version number, so when a minor change is made to a remote service, a new program number does not have to be assigned
- Whenever a new procedure is added to a program, it is also given an identifying number

When you want to call a procedure to find the number of remote users, you look up the appropriate program, version, and procedure numbers in a manual, just as you look up the name of a memory allocator when you want to allocate memory.

The callrpc Library Routine

The simplest way of making remote procedure calls is with the RPC library routine callrpc(). The arguments are:

argv(1)	The name of the remote server machine.
RUSERSPROG	The program.
RUSERSVERS	The version.
RUSERSPROC_NUM	The procedure number. Together with the program and version numbers, this defines the procedure to be called.
xdr_void	An XDR filter.
0	An argument to be encoded and passed to the remote procedure.
xdr_u_long	A filter for decoding the results returned by the remote procedure.
&nusers	A pointer to the place where the procedure's results are to be stored. Multiple arguments and results are handled by embedding them in structures.

If callrpc() completes successfully, it returns zero; otherwise, it returns a nonzero value. The return codes (cast into an integer) are found in clnt.h.

Since data types may be represented differently on different machines, callrpc() needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result).

For RUSERSPROC_NUM, the return value is an unsigned long so callrpc() has xdr_u_long() as its first return parameter, which says that the result is of type unsigned long and &nusers is its second return parameter, which is a pointer to where the long result is placed. Since RUSERSPROC_NUM takes no argument, the argument parameter of callrpc() is xdr_void().

If callrpc() gets no answer after several tries to deliver a message, it returns an error. The delivery mechanism is User Datagram Protocol (UDP). Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed later in this document.

The remote server procedure corresponding to the previous example might look like this:

```
char *
nuser(indata)
    char *indata;
{
    unsigned long nusers;

    .
    .  Code here to compute the number of users
    .  and place result in variable nusers.
    .
    return((char *)&nusers);
}
```

It takes one argument: a pointer to the input of the remote procedure call (ignored in the example), and it returns a pointer to the result.

Note: In the current version of C, character pointers are the generic pointers, so both the input argument and the return value are cast to (char *).

Registering RPC Calls

Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests.

In this example, there is only a single procedure to register, so the main body of the server looks like this:

```
#include <stdio.h>
#include <rpc.h>
#include <rusers.h> /* (not provided with RPC/XDR) */
char *nuser();

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM, nuser, xdr_void,
xdr_u_long);
    svc_run();
    fprintf(stderr, "error: svc_run returned!\n");
    exit(1);
}
```

Note: This assumes that a program nuser exists in the RPC library. It is not provided as part of the TCPaccess RPC/XDR.

registerrpc Arguments

The registerrpc() routine registers a C procedure as corresponding to a given RPC procedure number. It has the following arguments:

RUSERPROG	The program of the remote procedure to be registered
RUSERSVERS	The version of the remote procedure to be registered
RUSERSPROC_NUM	The procedure number of the remote procedure to be registered
nuser	The name of the local procedure that implements the remote procedure
xdr_void()	XDR filters for the remote procedure's arguments. Multiple arguments are passed as structures.
xdr_u_long()	XDR filters for the remote procedure's results. Multiple results are passed as structures.

Only the UDP transport mechanism can use registerrpc(); thus, it is always safe in conjunction with calls generated by callrpc().

Note: The UDP transport mechanism can only deal with arguments and results less than eight KB bytes in length.

After registering the local procedure, the server program's main procedure calls svc_run(), the RPC library's remote procedure dispatcher. This function calls the remote procedures in response to RPC call messages. The dispatcher takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

Note: Called remote procedures must have the results stored as static variables or external to the called procedure, so that the value is not lost when the procedure is exited.

Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 according to the following table:

Program Number	Assignment
0x0 - 0x1fffffff	Defined by Sun: Sun Microsystems administers this group of numbers, which should be identical for all Sun customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range.
0x20000000 - 0x3fffffff	Defined by user: This group is reserved for specific customer applications. This range is intended primarily for debugging new programs.
0x40000000 - 0x5fffffff	Transient: This group is reserved for applications that generate program numbers dynamically.
0x60000000 - 0x7fffffff	Reserved for future use; should not be used.
0x80000000 - 0x9fffffff	Reserved for future use; should not be used.
0xa0000000 - 0xbfffffff	Reserved for future use; should not be used.
0xc0000000 - 0xdfffffff	Reserved for future use; should not be used.
0xe0000000 - 0xffffffff	Reserved for future use; should not be used.

To register a protocol specification, or to obtain a complete list of registered programs, send a request by network mail to rpc@sun.com, or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Avenue
Mountain View, CA 94043

When registering a protocol specification, please include a compilable rpcgen ".x" file describing your protocol. You are given a unique program number in return. The RPC program numbers and protocol specifications of standard Sun RPC services can be found in the include files in /usr/include/rpcsvc on most UNIX machines. These services, however, constitute only a small subset of those that have been registered.

Passing Arbitrary Data Types

In the previous example, the RPC call passes a single unsigned long. RPC can handle arbitrary data structures, regardless of byte orders or structure layout conventions used by different machines. RPC does this by always converting the data structures to a network standard called External Data Representation (XDR) before sending them over the network.

The process of converting from a particular machine representation to XDR format is called serializing and the reverse process is called deserializing. The type field parameters of callrpc() and registerrpc() can be a built-in procedure such as xdr_u_long() in the previous example, or a user supplied one.

Built-in Type Routines

XDR has these built-in type routines:

```
xdr_int()
xdr_u_int()
xdr_enum()
xdr_long()
xdr_u_long()
xdr_bool()
xdr_short()
xdr_u_short()
xdr_wrapstring()
xdr_char()
xdr_u_char()
xdr_array()
xdr_bytes()
xdr_double()
xdr_float()
xdr_string()
xdr_union()
xdr_vector()
```

The routine xdr_string() exists, but cannot be used with callrpc() and registerrpc(), which only pass two parameters to their XDR routines. xdr_wrapstring, which has only two parameters, is syntactically correct, and it calls xdr_string().

Example 1

The following is an example of a user-defined type routine, if you want to send this structure:

```
struct simple
{
    int a;
    short b;
} simple;
```

you would call callrpc() by entering:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple, &simple ...);
```

Example 2

The xdr_simple() routine is written as follows:

```
#include <rpc.h>

xdr_simple(xdrsp, simplep)
    XDR *xdrsp;
    struct simple *simplep;
{
    if (!xdr_int(xdrsp, &simplep->a))
        return (0);
    if (!xdr_short(xdrsp, &simplep->b))
        return (0);
    return (1);
}
```

An XDR routine returns nonzero (true for C) if it completes successfully, and zero otherwise. You will find a complete description of XDR in RFC 1014; only a few implementation examples are given here.

Prefabricated Building Blocks

In addition to the built-in primitives, there are also the prefabricated building blocks:

```
xdr_array()
xdr_bytes()
xdr_reference()
xdr_vector()
xdr_union()
xdr_pointer()
xdr_string()
xdr_opaque()
```

To send a variable array of integers, you might package them as a structure:

```
struct varintarr
{
    int *data;
    int arrlnth;
}
arr;
```

Then make an RPC call such as:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_varintarr, &arr...);
```

The xdr_varintarr() routine is defined like this:

```
xdr_varintarr(xdrsp, arrp)
    XDR *xdrsp;
    struct varintarr *arrp;
{
    return
        (xdr_array(xdrsp, &arrp->data, &arrp->arrlnth, MAXLEN, sizeof(int),
xdr_int));
}
```

xdr_varintarr Arguments

The xdr_varintarr() routine takes the following arguments:

xdr_array	The XDR handle
&arrp->data	A pointer to the array
&arrp->arrlnth	A pointer to the size of the array
MAXLEN	The maximum allowable array size
sizeof(int)	The size of each array element
xdr_int	An XDR routine for handling each array element

If the size of the array is known in advance, you can use xdr_vector(), which serializes fixed-length.

```
int intarr[SIZE]; /* externally defined results */
xdr_intarr(xdrsp, intarr)
    XDR *xdrsp;
    int intarr[];
{
    int i;
    return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),            xdr_int));
}
```

XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the previous examples involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine xdr_bytes, which is like xdr_array except that it packs characters; xdr_bytes has four arguments, similar to the first four arguments of xdr_array. For null-terminated strings, there is also the xdr_string() routine, which is the same as xdr_bytes without the length argument. On select, it gets the string length from strlen, and on deserializing it creates a null-terminated string.

A Final Example

This is a final example that calls the previously written xdr_simple() as well as the built-in functions xdr_string() and xdr_reference (which chases pointers).

```
struct finalexample
{
    char *string;
    struct simple *simplep;
} finalexample;

xdr_finalexample(xdrsp, finalp)
    XDR *xdrsp;
    struct finalexample *finalp;
{
    if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
          return (0);
    if (!xdr_reference(xdrsp, &finalp->simplep,
                       sizeof(struct simple), xdr_simple);
          return (0);
    return (1);
}
```

Note: You could call xdr_simple() here instead of xdr_reference().

Lowest Layer of RPC

In the examples given in the previous section, RPC takes care of many details automatically. This section shows how to change the defaults by using lower layers of the RPC library. It is assumed that you are familiar with sockets, and with the system/function calls for dealing with them.

There are several occasions when you may need to use lower layers of RPC:

- You may need to use TCP, since the higher layer uses UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send long streams of data.
- You may want to allocate and free memory while serializing or deserializing with XDR routines. There is no call at the higher level to let you free memory explicitly. (See Memory Allocation with XDR additional information.)
- You may need to perform authentication on either the client or the server side, by supplying credentials or verifying them. See Authentication for additional information.

More on the Server Side

This server for the nusers program does the same thing as the one using registerrpc() shown earlier in this chapter, but is written using a lower layer of the RPC package:

```
#include <stdio.h>
#include <rpc.h>
#include <utmp.h>
#include <rusers.h> /* not provided with TCPaccess RPC/XDR */

#define RUSERSPROG 10002
#define RUSERSVERS 2
#define RUSERPROC-NUM 1

main()
{

    SVCXPRT *transp;
    int nuser();
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL)
    {
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser, IPPROTO_UDP))
    {
        fprintf(stderr, "can't register RUSER service\n");
        exit(1);
    }
    svc_run(); /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}
```

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;

  switch (rqstp->rq_proc)
  {
    case NULLPROC:
      if (!svc_sendreply(transp, xdr_void, 0))
        fprintf(stderr, "can't reply to RPC call\n");
      return;
    case RUSERSPROC_NUM:
      .
      . Code here to compute the number of users
      . and assign it to the variable nusers
      .
      if (!svc_sendreply(transp, xdr_u_long, &nusers))
        fprintf(stderr, "can't reply to RPC call\n");
      return;
    default:
      svcerr_noproc(transp);
      return;
  }
}
```

The Server Gets a Transport Handle

First, the server gets a transport handle, which is used for receiving and replying to RPC messages. registerrpc() uses svcudp_create to get a UDP handle. If you require a more reliable protocol, call svctcp_create instead. If the argument to svcudp_create is RPC_ANYSOCK, the RPC library creates a socket on which to receive and reply to RPC calls. Otherwise, svcudp_create expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of svcudp_create and clnttcp_create (the low-level client routine) must match.

If the user specifies the RPC_ANYSOCK argument, the RPC library routines open a socket. Otherwise, they expect the user to do so. The routines svcudp_create and clntudp_create cause the RPC library routines to bind their sockets, if they are not bound already.

A service may choose to register its port number with the local portmapper service. This is done by specifying a non-zero protocol number in svc_register. A client can discover the server's port number by consulting the portmapper on their server's machine. This can be done automatically by specifying a zero port number in clntudp_create or clnttcp_create.

The Server Calls pmap_unset

After creating an SVCXPRT, the next step is to call pmap_unset so that if the nusers server crashed earlier, any previous trace of it is erased before restarting. More precisely, pmap_unset erases the entry for RUSERSPROG from the portmapper's tables.

The Program Number is Associated with the nuser Procedure

Finally, the program number for nusers is associated with the procedure nuser. The final argument to svc_register is normally the protocol in use, which, in this case, is IPPROTO_UDP. Unlike registerrpc(), there are no XDR routines involved in the registration process. Also, registration is done on the program level, rather than the procedure level.

The user routine nuser must call and dispatch the appropriate XDR routines based on the procedure number. These tasks, which registerrpc() handles automatically, are handled by nuser:

- Procedure NULLPROC (currently zero) returns with no results. This can be used as a simple test to detect if a remote program is running.
- There is a check for invalid procedure numbers. If one is detected, svcerr_noproc is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via svc_sendreply. Its first argument is the SVCXPRT handle, the second is the XDR routine, and the third is a pointer to the data to be returned.

Handling an RPC Program that Receives Data

Not illustrated in the previous example is how a server handles an RPC program that receives data. You can add a procedure RUSERSPROC_BOOL, which has an argument nusers, and returns TRUE or FALSE depending on whether there are nusers logged on. This is an example of how it looks:

```
case RUSERSPROC_BOOL:
{
    int bool;
    unsigned nuserquery;

    if (!svc_getargs(transp, xdr_u_int, &nuserquery)
    {
        svcerr_decode(transp);
        return;
    }
```

```
.
.  Code to set nusers = number of users
.
if (nuserquery == nusers)
    bool = TRUE;
else
    bool = FALSE;
if (!svc_sendreply(transp, xdr_bool, &bool))
    {
        fprintf(stderr, "can't reply to RPC call\n");
        return (1);
    }
return;
}
```

The relevant routine is svc_getargs, which takes, as arguments, an SVCXPRT handle, the XDR routine, and a pointer to where the input is to be placed.

Memory Allocation with XDR

XDR routines not only process input and output, they also perform memory allocation. This is why the second argument of xdr_array is a pointer to an array, rather than the array itself. If it is NULL, then xdr_array allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider this XDR routine xdr_chararr1, which deals with a fixed array of bytes with length SIZE.

```
xdr_chararr1(xdrsp, chararr)
    XDR *xdrsp;
    char chararr[];
{
    char *p;
    int len;

    p = chararr;
    len = SIZE;
    return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

If space has already been allocated in chararr, it can be called from a server like this:

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

If you want XDR to do the allocation, you would have to rewrite the routine like this:

```
xdr_chararr2(xdrsp, chararrp)
    XDR *xdrsp;
    char **chararrp;
{
    int len;

    len = SIZE;
    return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

Then the RPC call might look like this:

```
char *arrptr;

arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);

/* Use the result here */

svc_freeargs(transp, xdr_chararr2, &arrptr);
```

After being used, the character array can be freed with svc_freeargs. svc_freeargs does not attempt to free any memory if the variable indicating it is NULL.

Note: In the routine xdr_finalexample, given earlier, if finalp->string was NULL, then it would not be freed. The same is true for finalp->simplep.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from callrpc(), the serializer part is used. When called from svc_getargs, the deserializer is used. And when called from svc_freeargs, the memory deallocator is used.

When building simple examples like those in this section, a user does not have to worry about the three modes. See RFC 1014 for examples of more sophisticated XDR routines that determine which of the three modes they are in and adjust their behavior accordingly.

The Calling Side

When you use callrpc(), you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the layer of RPC that lets you adjust these parameters, consider this code to call the nusers service. This program, as shown, can be run on MVS.

```
#include <stdio.h>
#include <rpc.h>
#include <utmp.h>
#include <netdb.h>
#define RUSERSPROG 10002
#define RUSERSVERS 2
#define RUSERSPROC-NUM 1
main(argc, argv)
    int argc; char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;
    if (argc != 2)
    {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
```

```
    }
    if ((hp = gethostbyname(argv[1])) == NULL)
    {
        fprintf(stderr, "can't get addr for %s\n",argv[1]);
        exit(-1);
    }
    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = 0;
    if ((client = clntudp_create(&server_addr, RUSERSPROG,
                                 RUSERSVERS, pertry_timeout, &sock)) == NULL)
        {
            clnt_pcreateerror("clntudp_create");
            exit(-1);
        }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
                          0, xdr_u_long, (char*)&nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS)
    {
        clnt_perror(client, "rpc");
        exit(-1);
    }
    clnt_destroy(client);
    close(sock);
    exit(0);
}
```

The CLIENT Pointer

The low-level version of callrpc() is clnt_call(), which takes a CLIENT pointer rather than a host name. It takes the following arguments:

client	A CLIENT pointer
RUSRSPROC_NUM	The procedure number
xdr_void	The XDR routine for serializing the argument
0	A pointer to the argument
xdr_u_long	The XDR routine for deserializing the return value
(char*)&nusers	A pointer to where the return value is placed
total_timeout	The total time, in seconds, to wait for a reply

The CLIENT pointer is encoded with the transport mechanism. The callrpc() routine uses UDP, thus it calls clntudp_create to get a CLIENT pointer. To get Transmission Control Protocol (TCP), you use clnttcp_create.

The clntudp_create() Arguments

The clntudp_create() routine takes five arguments; in this example, they are:

&server_addr	The server address
RUSERSPROG	The program number
RUSERSVERS	The version number
pertry_timeout	A timeout value (between tries)
&sock	A pointer to a socket

Thus, the number of tries is the clnt_call() timeout divided by the clntudp_create() timeout.

The clnt_destroy call always deallocates the space associated with the CLIENT handle. It closes the socket associated with the CLIENT handle, however, only if the RPC library opened it.

If the socket was opened by the user, it stays open. This makes it possible, in cases where multiple client handles are using the same socket, to destroy one handle without closing the socket that other handles are using.

To make a stream connection, the call to clntudp_create is replaced with a call to clnttcp_create.

```
clnttcp_create(&server_addr, prognum, versnum, &sock, inputsize, outputsize);
```

There is no timeout argument; instead, the receive and send buffer sizes must be specified. When the clnttcp_create call is made, a connection is established. All RPC calls using that CLIENT handle would use this connection. The server side of an RPC call using TCP has svcudp_create replaced by svctcp_create.

```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
```

The last two arguments to svctcp_create are send and receive sizes, respectively. If 0 is specified for either of these, the system chooses a reasonable default.

Useful RPC Features

This section discusses some other aspects of RPC that you may find useful.

Select on the Server Side

If a process is handling RPC requests while performing some other activity, and the other activity involves periodically updating a data structure, the process can set an alarm signal before calling svc_run(). But if the other activity involves waiting on a file descriptor, the svc_run() call will not work.

The code for svc_run() should be like the following:

```
void
svc_run()
{
    fd_set readfds;
    int dtbsz = getdtablesize();

/* Note: getdtablesize is not provided with RPC/XDR    */
    for (;;)
    {
        readfds = svc_fds;
        switch (select(dtbsz, &readfds, NULL,NULL,NULL))
        {
            case -1:
                if (errno == EINTR)
                    continue;
                perror("select");
                return;
            case 0:
                break;
            default:
                svc_getreqset(&readfds);
        }
    }
}
```

You can bypass svc_run() and call svc_getreqset yourself. All you need to know are the file descriptors of the sockets or sockets associated with the programs you are waiting on. Thus, you can have your own select that waits on both the RPC socket and your own descriptors. svc_fds is a bit mask of all the file descriptors that RPC is using for services. It can change every time that any RPC library routine is called, because descriptors are constantly being opened and closed (e.g., for connections).

For users who prefer to use generic ECBs for synchronization, mvs_svc_run() may be used. In this case, the RPC server acts the same as if called using svc_run(), but control returns to the caller of mvs_svc_run() when an ECB is posted.

Broadcast RPC

The portmapper is a daemon that converts RPC program numbers into DARPA protocol port numbers. For more information about the portmapper, read RFC 1057.

Note: Broadcast RPC is not available in the TCPaccess RPC/XDR.

You cannot broadcast RPC without the portmapper. Here are the main differences between broadcast RPC and normal RPC calls:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
- Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UDP/IP.
- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
- All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.
- Broadcast requests are limited in size to the Maximum Transfer Unit (MTU) of the local network. For Ethernet, the MTU is 1500 bytes.

Broadcast RPC Synopsis

```
#include <pmapclnt.h>
    .
    .
    .
enum clnt_stat clnt_stat;
    .
    .
    .
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
    inproc, in, outproc, out, eachresult)
        u_long prognum;          /* program number */
        u_long versnum;          /* version number */
        u_long procnum;          /* procedure number */
        xdrproc_t inproc;        /* xdr routine for args */
        caddr_t in;              /* pointer to args */
        xdrproc_t outproc;       /* xdr routine for results */
        caddr_t out;             /* pointer to results */

        bool_t (*eachresult)();  /* call with each result */
```

The procedure is called each time a valid result is obtained. It returns a boolean that indicates whether the user wants more responses.

```
bool_t done;
    .
    .
    .
done = eachresult(resultsp, raddr)
    caddr_t resultsp;
    struct sockaddr_in *raddr;   /* Addr of responding machine */
```

If done is TRUE, then broadcasting stops and clnt_broadcast returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with RPC_TIMEDOUT.

Batching

In the RPC architecture, clients send a call message and wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgment for every message sent. It is possible for clients to continue computing while waiting for a response using RPC batch facilities.

Server Batching

RPC messages can be placed in a *pipeline* of calls to a desired server; this is called batching. Batching assumes that:

- Each RPC call in the pipeline requires no response from the server, and the server does not send a response message; and
- The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP.

Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages, and send them to the server in one write system call. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes, as well as the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually do a nonbatched call to flush the pipeline.

Here is an example of batching. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent.

The service (using the TCP/IP transport) may look like this:

```
#include <stdio.h>
#include <rpc.h>

void windowdispatch();
main()
{
    SVCXPRT *transp;
    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL)
    {
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(WINDOWPROG, WINDOWVERS);
    if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
                      windowdispatch, IPPROTO_))
    {
        fprintf(stderr, "can't register WINDOW service\n");
        exit(1);
    }
    svc_run();                  /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}
void windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;

{
    char *s = NULL;
    switch (rqstp->rq_proc)
    {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
                fprintf(stderr, "can't reply to RPC call\n" );
            return;
        case RENDERSTRING:
            if (!svc_getargs(transp, xdr_wrapstring, &s))
            {
                fprintf(stderr, "can't decode arguments\n") ;

                /* Tell caller about error */
                svcerr_decode(transp);
                break;
            }
            .
            .  Code here to render the strings
            .
            if (!svc_sendreply(transp, xdr_void, NULL))
                fprintf(stderr, "can't reply to RPC call\n" );
            break;
        case RENDERSTRING_BATCHED:
            if (!svc_getargs(transp, xdr_wrapstring, &s))
            {
                fprintf(stderr, "can't decode arguments\n") ;

                /* We are silent in the face of protocol errors */
```

```
            break;
            }
        .
        .  Code here to render strings, but send no reply!
        .
        break;
        default:
            svcerr_noproc(transp);
            return; }

        /* Now free string allocated while decoding arguments */

        svc_freeargs(transp, xdr_wrapstring, &s);
    }
}
```

The service could have one procedure that takes the string and a boolean to indicate whether the procedure should respond.

Client Batching

For a client to take advantage of batching, the client must perform RPC calls on a TCP/IP-based transport and the actual calls must have these attributes:

- The resulting XDR routine must be zero (NULL).
- The RPC call's timeout must be zero.

Here is an example of a client that uses batching to render multiple strings; the batching is flushed when the client gets a null string (EOF):

```
#include <stdio.h>
#include <rpc.h>
#include <socket.h>
#include <time.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    char buf[1000], *s = buf;


   if ((client = clnttcp_create(&server_addr, WINDOWPROG,
                      WINDOWVERS, &sock, 0, 0)) == NULL)
    {
        error("clnttcp_create");
        exit(-1);
    }
    total_timeout.tv_sec = 0;
    total_timeout.tv_usec = 0;
    while (scanf("%s", s) != EOF)
   {
            clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
```

```
            xdr_wrapstring, &s, NULL, NULL, total_timeout);
             if (clnt_stat != RPC_SUCCESS)
             {
                 clnt_perror(client, "batched rpc");
                 exit(-1);
             }
    }

    /* Now flush the pipeline */

    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
        xdr_void, NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS)
    {
        clnt_perror(client, "rpc");
        exit(-1);
    }
    clnt_destroy(client);
    exit(0);
}
```

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

The previous example was completed to render all of the (2000) lines in the UNIX file /etc/termcap. The rendering service did nothing but throw the lines away. The example was run in the following configurations, with the indicated results:

Configuration	Result
Machine to itself, regular RPC	50 seconds
Machine to itself, batched RPC	16 seconds
Machine to machine, regular RPC	52 seconds
Machine to machine, batched RPC	10 seconds

Running fscanf on the UNIX file /etc/termcap only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, though these protocols are often hard to design.

Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network file system, require stronger security than what has been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type is none.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support.

UNIX Authentication

The TCPaccess RPC/XDR supports UNIX authentication.

The Client Side

When a caller creates a new RPC client handle by using:

```
clnt = clntudp_create(address, prognum, versnum, wait, sockp)
```

the appropriate transport instance defaults the associate authentication handle to:

```
clnt->serverl_auth = authnone_create()
```

The RPC client can choose to use UNIX style authentication by setting the field clnt->serverl_auth after creating the RPC client handle using:

```
clnt->serverl_auth = authunix_create_default()
```

This causes each RPC call associated with clnt to carry an authentication credentials structure:

```
.
. UNIX style credentials
.
struct authunix_parms
{
    u_long   aup_time;          /* credentials creation time */
    char    *aup_machname;      /* host name where client is */
    int      aup_uid;           /* client's UNIX effective uid */
    int      aup_gid;           /* client's current group id */
    u_int    aup_len;           /* element length of aup_gids */
    int     *aup_gids;          /* array of groups user is in */
};
```

These fields are set by authunix_create_default by using the appropriate system calls. Since the RPC user created this new style of authentication, the user is responsible for destroying it with

```
auth_destroy(clnt->serverl_auth);
```

This should be done in all cases to conserve memory.

The Server Side

Service implementers have a harder time dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine using this code:

```
.
.  An RPC Service request
.
struct svc_req
{
    u_long  rq_prog;              /* service program number */
    u_long  rq_vers;              /* service protocol vers num */
    u_long  rq_proc;              /* desired procedure number */
    struct  opaque_auth rq_cred;  /* raw credentials from wire */
    caddr_t rq_clntcred;          /* credentials (read only) */
};
```

The rq_cred is mostly opaque, except for one field of interest: the style or flavor of authentication credentials, as illustrated here:

```
.
.  Authentication info, mostly opaque to the programmer
.
struct opaque_auth
{
    enum_t oa_flavor;           /* style of credentials */
    caddr_t oa_base;            /* address of more auth stuff */
    u_int oa_length;            /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package makes these guarantees to the service dispatch routine:

- The request's rq_cred is well formed. Thus, the service implementer may inspect the request's rq_cred.oa_flavor to determine which style of authentication the caller used. The service implementer may also wish to inspect the other fields of rq_cred if the style is not one of the styles supported by the RPC package.
- The request's rq_clntcred field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. Remember that only UNIX style is currently supported, so (currently) rq_clntcred could be cast to a pointer to an authunix_parms structure. If rq_clntcred is NULL, the service implementer may wish to inspect the other (opaque) fields of rq_cred, in case the service knows about a new type of authentication that the RPC package does not know about.

This remote users service example can be extended so that it computes results for all users except UID 16.

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authunix_parms *unix_cred;
    int uid;
    unsigned long nusers;   /* we don't care about authentication
                               for null proc */
    if (rqstp->rq_proc == NULLPROC)
    {
        if (!svc_sendreply(transp, xdr_void, 0))
        {
            fprintf(stderr, "can't reply to RPC call\n" );
            return (1);
        }
        return;
    }
/* now get the uid */

    switch (rqstp->rq_cred.oa_flavor)
    {
        case AUTH_UNIX:
          unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
          uid = unix_cred->aup_uid;
          break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
    }
    switch (rqstp->rq_proc)
    {
        case RUSERSPROC_NUM:   /* make sure caller is allowed
                                  to call this proc */
            if (uid == 16)
            {
                svcerr_systemerr(transp);
                return;
            }
            .
            .  Code here to compute the number of users
            .  and assign it to the variable nusers
            .
            if (!svc_sendreply(transp, xdr_u_long, &nusers))
            {
                fprintf(stderr, "can't reply to RPC call\n" );
                return (1);
            }
            return;
        default:
            svcerr_noproc(transp);
            return;
    }
}
```

A few things should be noted:

- It is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero).
- If the authentication parameter's type is not suitable for your service, you should call svcerr_weakauth.
- The service protocol itself should return status for access denied; in the case of the example, the protocol does not have such a status, so the service primitive svcerr_systemerr is called instead.
- The last point underscores the relation between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

DES Authentication

UNIX authentication is quite easy to defeat. Instead of using authunix_create_default, you can call authunix_create and then modify the RPC authentication handle it returns by filling in whatever user ID and host name you want the server to think it has. DES authentication is thus recommended if you want more security than UNIX authentication offers.

Note: The TCPaccess RPC/XDR does not currently support DES authentication. Users may write their own DES authorization handler.

The details of the DES authentication protocol are complicated and are not explained here. See RFC 1057 for the details.

For DES authentication to work, the keyserv(8c) daemon must be running on both the server and client machines. The users on these machines need public keys assigned by the network administrator in the publickey(5) database. They also need to have decrypted their secret keys using their login password. This happens automatically when you log in using login(1), or you can do it manually using keylogin(1).

Client Side

If a client wishes to use DES authentication, it must set its authentication handle appropriately. Here is an example:

```
cl->cl-auth=
    authdes_create(servername, 60, &server_addr, NULL);
```

The first argument is the network name or *netname* of the owner of the server process. Typically, server processes are root processes and their netname can be derived using this call:

```
char servername[MAXNETNAMELEN];

host2netname(servername, rhostname, NULL);
```

Here, rhostname is the host name of the machine where the server process is running. host2netname fills in servername to contain this root process's netname. If the server process was run by a regular user, you could use the call user2netname instead. Here is an example for a server process with the same user ID as the client:

```
char servername[MAXNETNAMELEN];

user2netname(servername, getuid(), NULL);
```

The last argument to both user2netname and host2netname is the name of the naming domain where the server is located. The NULL used here means "use the local domain name".

authdes_create Arguments

The authdes_create routine takes the following arguments:

servername	The name of the server
60	The lifetime of the credential Here it is set to sixty seconds. This means that the credential expires 60 seconds from now. If some mischievous user tries to reuse the credential, the server RPC subsystem recognizes that it has expired and will not grant any requests. If the same user tries to reuse the credential within the sixty-second lifetime, the user is still rejected because the server RPC subsystem remembers which credentials it has already seen in the near past, and does not grant requests to duplicates.
&server_addr	The address of the host to which it synchronizes / For DES authentication to work, the server and client must agree on the time. Here the address of the server itself is passed, so the client and server are both using the same time: the server's time. The argument can be NULL, which means "don't bother synchronizing". You should only do this if you are sure the client and server are already synchronized.

NULL	The address of a DES encryption key to use for encrypting time stamps and data / If this argument is NULL, as it is in this example, a random key is chosen. The client may find out the encryption key being used by consulting the ah_key field of the authentication handle.

Server Side

The server side is a lot simpler than the client side. Here is the previous example rewritten to use AUTH_DES instead of AUTH_UNIX:

```
#include <time.h>
#include <authdes.h>
.
.
.
nuser(rqstp, transp)
    struct svc_req *rqstp;

    SVCXPRT *transp;
{
    struct authdes_cred *des_cred;
    int uid;
    int gid;
    int gidlen;
    int gidlist[10];[
/* we don't care about authentication for null proc */
    if (rqstp->rq_proc == NULLPROC)
    {
        .
        .  same as before
        .
    }
/* now get the uid     */
    switch (rqstp->rq_cred.oa_flavor)
    {
        case AUTH_DES:
        des_cred =
          (struct authdes_cred *) rqstp->rq_clntcred;
        if (! netname2user(des_cred->adc_fullname.name, &uid,
             &gid, &gidlen, gidlist))
        {
            fprintf(stderr, "unknown user: %s", des_cred->adc_fullname.name);
            svcerr_systemerr(transp);
            return;
        }
        break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
    }
        .
        .  The rest is the same as before
        .
}
```

Notice the use of the routine netname2user, the inverse of user2netname: it takes a network ID and converts to a UNIX ID. netname2user also supplies the group IDs that are not used in this example, but which may be useful to other UNIX programs.

Using Inetd

Note: inetd is not available in the TCPaccess RPC/XDR.

An RPC server can be started from inetd.

When starting an RPC server from inetd, the only difference from the usual code is that the service creation routine should be called in this form, since inet passes a socket as file descriptor 0.

```
transp = svcudp_create(0);       /* For UDP */
transp = svctcp_create(0,0,0);   /* For listener tcp sockets */
transp = svcfd_create(0,0,0);    /* For connected tcp sockets */
```

Also, svc_register should be called with the final flag as 0, since the program would already be registered by inetd.

```
svc_register(transp, PROGNUM, VERSNUM, service, 0);
```

Remember, if you want to exit from the server process and return control to inet, you need to explicitly exit, since svc_run() never returns.

The format of entries in /etc/inetd.conf for RPC services can be either one of these:

```
p_name/version dgram rpc/udp wait/nowait user server args p_name/version stream
rpc/tcp wait/nowait user server args
```

p_name	The symbolic name of the program as it appears in rpc(5).
server	The program implementing the server.
version	The version number of the service.

If the same program handles multiple versions, then the version number can be a range, as in this example:

```
rstatd/1-2 dgram rpc/udp wait root /usr/etc/rpc.rstatd
```

Programming Examples

This section presents more examples of remote procedure calls.

Versions

By convention, the first version number of program PROG is PROGVERS_ORIG; the most recent version is PROGVERS. For a new version of the user program named RUSERSVERS_SHORT that returns an unsigned short rather than a long, a server that wants to support both versions would do a double register.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG, nuser,
                  IPPROTO_tcp))
{
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT, user,
                  IPPROTO_tcp))
{
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
```

Both versions can be handled by the same C procedure, as this example illustrates:

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2;

    switch (rqstp->rq_proc)
    {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
            {
                fprintf(stderr, "can't reply to RPC call\n");
                return (1);
            }
            return;
        case RUSERSPROC_NUM:
        .
        .   Code here to compute the number of users
        .   and assign it to the variable nusers
        .
            nusers2 = nusers;
            switch (rqstp->rq_vers)
            {
                case RUSERSVERS_ORIG:
                    if (!svc_sendreply(transp, xdr_u_long,
                        &nusers))
                    {
                      fprintf(stderr,"can't reply to RPC call\n");
                    }
                    break;
                case RUSERSVERS_SHORT:
                    if (!svc_sendreply(transp, xdr_u_short,
                          &nusers2))
```

```
                    {
                      fprintf(stderr,"can't reply to RPC call\n");
                    }
                    break;
                }
                default:
                    svcerr_noproc(transp);
                    return;
            }
}
```

TCP

Here is an example that is essentially rcp, a UNIX remote file copy program that copies files between machines. The initiator of the RPC snd call takes its standard input and sends it to the server rcv, which prints it on standard output. The RPC call uses TCP. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

```
/* The xdr routine: on decode, read from wire, write onto fp
 * on encode, read from fp, write onto wire */

#include <stdio.h>
#include <rpc.h>

xdr_rcp(xdrs, fp)
    XDR *xdrs; FILE *fp;
{
    unsigned long size;
    char buf[BUFSIZ], *p;

    if (xdrs->x_op == XDR_FREE)

    /* nothing to free */

        return 1;
    while (1)
    {
        if (xdrs->x_op == XDR_ENCODE)
        {
            if ((size = fread(buf, sizeof(char), BUFSIZ, fp)) == 0
                        && ferror(fp))
            {
                fprintf(stderr, "can't fread\n");
                return (1);
            }
        }
        p = buf;
        if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
            return 0;
        if (size == 0)
            return 1;
        if (xdrs->x_op == XDR_DECODE)
        {
            if (fwrite(buf, sizeof(char), size, fp) != size)
                {
                    fprintf(stderr, "can't fwrite\n");
                    return (1);
                }
        }
    }
}
```

```
/* The sender routines */
#include <stdio.h>
#include <netdb.h>
#include <rpc.h>
#include <socket.h>
#include <time.h>
main(argc, argv)
    int argc;
    char **argv;
{
    int xdr_rcp();
    int err;

    if (argc < 2)
    {
        fprintf(stderr, "usage: %s servername\n", argv[0]);
        exit(-1);
    }
    if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC,
                RCPVERS, xdr_rcp, stdin, xdr_void, 0) != 0))
        {
            clnt_perrno(err);
            fprintf(stderr, "can't make RPC call\n");
            exit(1);
        }
        exit(0);
}

callrpctcp(host, prognum, procnum, versnum, inproc,
           in, outproc, out)
    char *host, *in, *out;
    xdrproc_t inproc, outproc;
{
    struct sockaddr_in server_addr;
    int sockets, = RPC_ANYSOCK;
    enum clnt_stat clnt_stat;
    struct hostent *hp;
    register CLIENT *client;
    struct timeval total_timeout;

    if ((hp = gethostbyname(host)) == NULL)
    {
        fprintf(stderr, "can't get addr for '%s'\n", host);
        return (-1);
    }
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
            hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = 0;
    if ((client = clnttcp_create(&server_addr, prognum,
         versnum, &sockets,, BUFSIZ, BUFSIZ)) == NULL)
    {
        perror("rpctcp_create");
        return (-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum, inproc, in, outproc,
                out, total_timeout);
    clnt_destroy(client);
    return (int)clnt_stat;
}

/* The receiving routines     */
```

```
#include <stdio.h>
#include <rpc.h>

main()
{
    register SVCXPRT *transp;
        int rcp_service(), xdr_rcp();
    if ((transp = svctcp_create(RPC_ANYSOCK, BUFSIZ, BUFSIZ)) ==
                        NULL)
    {
        fprintf("svctcp_create: error\n");
        exit(1);
    }
    pmap_unset(RCPPROG, RCPVERS);
    if (!svc_register(transp, RCPPROG, RCPVERS, rcp_service, IPPROTO_tcp))
    {
        fprintf(stderr, "svc_register: error\n");
        exit(1);
    }
    svc_run(); /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}

rcp_service(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc)
    {
        case NULLPROC:
            if (svc_sendreply(transp, xdr_void, 0) == 0)
            {
                fprintf(stderr, "err: rcp_service");
                return (1);
            }
            return;
        case RCPPROC_FP:
            if (!svc_getargs(transp, xdr_rcp, stdout))
            {
                svcerr_decode(transp);
                return;
            }
            if (!svc_sendreply(transp, xdr_void, 0))
            {
                fprintf(stderr, "can't reply\n");
                return;
            }
            return (0);
        default:
            svcerr_noproc(transp);
            return;
    }
}
```

Callback Procedures

Occasionally, it is useful to have a server become a client and make an RPC call back to the process that is its client. An example is remote debugging, where the client is a window system program and the server is a debugger running on the remote machine. Most of the time the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed, and the debugger wants to make an RPC call to the window program, so that it can inform the user that a breakpoint has been reached.

To do an RPC callback, you need a program number on which to make the RPC call. Since this is a dynamically generated program number, it should be in the transient range, 0x40000000 - 0x5fffffff. The routine gettransient returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the gettransient routine itself. The call to pmap_set is a test and set operation, because it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the sockp argument points to a socket, that can be used as the argument to an svcudp_create() or svctcp_create() call.

```
#include <stdio.h>
#include <rpc.h>
#include <sockets.h>

gettransient(proto, vers, sockp)
    int proto, vers, *sockp;
{
    static int prognum = 0x40000000;
    int s, len, socktype;
    struct sockaddr_in addr;
    switch(proto)
    {
        case IPPROTO_UDP:
            socktype = SOCK_DGRAM;
            break;
        case IPPROTO_TCP:
            socktype = SOCK_STREAM;
            break;
        default:
            fprintf(stderr, "unknown protocol type\n");
            return 0;
    }
if (*sockp == RPC_ANYSOCK)
    {
        if ((s = sockets,(AF_INET, socktype, 0)) < 0)
        {
            perror("sockets,");
            return (0);
        }
        *sockp = s;
    }
else
    s = *sockp;
    addr.sin_addr.s_addr = 0;
    addr.sin_family = AF_INET;
    addr.sin_port = 0;
    len = sizeof(addr);
```

```
    /* may be already bound, so don't check for error */
    bind(s, &addr, len);
    if (getsockname(s, &addr, &len)< 0)
    {
        perror("getsockname");
        return (0);
    }
    while (!pmap_set(prognum++, vers, proto, ntohs(addr.sin_port))) continue;
    return (prognum-1);
}
```

Note: The call to ntohs is necessary to ensure that the port number in addr.sin_port, which is in network byte order, is passed in host byte order (as pmap_set expects).

The following client and server programs illustrate how to use the gettransient routine. The client makes an RPC call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the program EXAMPLEPROG, so that it can receive the RPC call informing it of the callback program number. Then at some random time (on receiving an ALRM signal in this example), it sends a callback RPC call, using the program number it received earlier.

Client

The client program:

```
/* client    */

#include <stdio.h>
#include <rpc.h>

int callback,();
char hostname[256];
main()

{
    int x, ans, s;
    SVCXPRT *xprt;
    gethostname(hostname, sizeof(hostname));

    s = RPC_ANYSOCK;
    x = gettransient(IPPROTO_UDP, 1, &s);
    fprintf(stderr, "client gets prognum %d\n", x);
    if ((xprt = svcudp_create(s)) == NULL)
    {
        fprintf(stderr, "rpc_server: svcudp_create\n");
        exit(1);
    }
    /* protocol is 0 - gettransient does registering */

    (void)svc_register(xprt, x, 1, callback,, 0);
    ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
                  EXAMPLEPROC_callback, xdr_int, &x, xdr_void, 0);
    if ((enum clnt_stat) ans != RPC_SUCCESS)
    {
        fprintf(stderr, "call: ");
        clnt_perrno(ans);
        fprintf(stderr, "\n");
    }
    svc_run();
    fprintf(stderr, "error: svc_run shouldn't return\n");
}
callback,(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc)
    {
        case 0:
            if (!svc_sendreply(transp, xdr_void, 0))
            {
                fprintf(stderr, "err: exampleprog\n ");
                return (1);
            }
            return (0);
        case 1:
            if (!svc_getargs(transp, xdr_void, 0))
            {
                svcerr_decode(transp);
                return (1);
            }
            fprintf(stderr, "client got callback,\n");
            if (!svc_sendreply(transp, xdr_void, 0))
            {
                fprintf(stderr, "err: exampleprog") ;
                return (1);
            }
    }
}
```

Server

The server program:

```
/* server     */
#include <stdio.h>
#include <rpc.h>
#include <signal.h>

char *getnewprog();
char hostname[256];
int docallback,();
int pnum;             /* program number for callback, routine */

main()
{
    gethostname(hostname, sizeof(hostname));
    registerrpc(EXAMPLEPROG, EXAMPLEVERS,
                EXAMPLEPROC_callback,, getnewprog, xdr_int, xdr_void);
    fprintf(stderr, "server going into svc_run\n");
    signal(SIGALRM, docallback,);
    alarm(10);
    svc_run();
    fprintf(stderr, "error: svc_run shouldn't return\n");
}
char *
getnewprog(pnump)
    char *pnump;
{
    pnum = *(int *)pnump;
    return NULL;

}
docallback,()
{
    int ans;

    ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0, xdr_void, 0);
    if (ans != 0)
    {
         fprintf(stderr, "server: ");
         clnt_perrno(ans);
         fprintf(stderr, "\n");
    }
}
```

Chapter 3

XDR: Technical Notes

This chapter contains technical notes on this implementation of the External Data Representation (XDR) standard, a set of library routines that enable a C programmer to describe arbitrary data structures in a machine-independent fashion.

It includes these sections:

- XDR Library Primitives—Provides a synopsis of each XDR primitive
- Advanced Topics—Describes additional techniques for passing data structures

For a formal specification of the XDR standard, read RFC 1014. XDR is the backbone of the Remote Procedure Call package, in the sense that data for remote procedure calls is transmitted using the standard. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.

This chapter contains a short tutorial overview of the XDR library routines, a guide to accessing currently available XDR streams, and information on defining new streams and data types.

XDR was designed to work across different languages, operating systems, and machine architectures.

Most users (particularly RPC users) only need this information:

- Number Filters
- Floating Point Filters
- Enumeration Filters

If you want to implement RPC and XDR on new machines, read the rest of this chapter, as well as RFCs 1014 and 1057, which are your primary references.

Note: You can use rpcgen to write XDR routines even in cases where no RPC calls are being made.

Justification

Consider the following writer and reader programs.

Writer

```
#include <stdio.h>
main() /* writer.c */
{
    long i;

    for (i = 0; i < 8; i++)
    {
        if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1)
        {
            fprintf(stderr, "failed!\n"); exit(1);
        }
    }
    exit(0);
}
```

Reader

```
#include <stdio.h>
main() /* reader.c */
{
    long i, j;

    for (j = 0; j < 8; j++)
    {
        if (fread((char *)&i, sizeof (i), 1, stdin) != 1)
        {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

Execution Results

The two programs appear to be portable for these reasons:

- They pass lint checking
- They exhibit the same behavior when executed on two different hardware architectures, a Sun and a VAX

Piping the output of the writer program to the reader program gives identical results on a Sun or a VAX.

```
sun% writer | reader
3.0.1.1 1 2 3 4 5 6 7
sun%

vax% writer | reader
3.0.1.2 1 2 3 4 5 6 7
vax%
```

Network Pipes

With the advent of local area networks and 4.2BSD came the concept of *network pipes* – a that process produces data on one machine, and a second process consumes data on another machine. A network pipe can be constructed with writer and reader.

Here are the results if the first produces data on a Sun, and the second consumes data on a VAX.

```
sun% writer | rsh vax reader
3.0.1.3 16777216 33554432 50331648 67108864 83886080 100663296 117440512
sun%
```

Identical results can be obtained by executing writer on the VAX and reader on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, even though the word size is the same.

For example, 16777216 is 224 – when four bytes are reversed, the one ends up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the read() and write() calls with calls to an XDR library routine xdr_long() a filter that knows the standard representation of a long integer in its external form.

Revised Writer

The following is the revised version of writer:

```
#include <stdio.h>
#include <rpc.h>                /* xdr is a sub-library of rpc */

main()                          /* writer.c */
{
    XDR xdrs;
    long i;

    xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
    for (i = 0; i < 8; i++)
    {
        if (!xdr_long(&xdrs, &i))
        {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

Revised Reader

Here is the revised version of reader:

```
#include <stdio.h>
#include <rpc.h>                /* xdr is a sub-library of rpc */
main()                          /* reader.c */
{
    XDR xdrs;
    long i, j;

    xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (j = 0; j < 8; j++)
    {
        if (!xdr_long(&xdrs, &i))
        {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

Revised Execution Results

The new programs were executed on a Sun, on a VAX, and from a Sun to a VAX with these results:

```
sun% writer | reader
3.0.1.4 1 2 3 4 5 6 7
sun%

vax% writer | reader
3.0.1.5 1 2 3 4 5 6 7
vax%

sun% writer | rsh vax reader
3.0.1.6 1 2 3 4 5 6 7
sun%
```

Integers are just the tip of the portable-data iceberg. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. And pointers, which are very convenient to use, have no meaning outside the machine where they are defined.

A Canonical Standard

XDR's approach to standardizing data representations is canonical. That is, XDR defines a single byte order (big-endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations; similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents.

The single standard completely de-couples programs that create or send portable data from those that use or receive portable data. The advent of a new machine or a new language has no effect on the community of existing portable data creators and users. A new machine joins this community by being *taught* how to convert the standard representations and its local representations; the local representations of other machines are irrelevant. Conversely, to existing programs running on other machines, the local representations of the new machine are also irrelevant; such programs can immediately read portable data produced by the new machine because such data conforms to the canonical standards that they already understand.

There are strong precedents for XDR's canonical approach (for example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols). The advantage of any canonical approach is simplicity; in the case of XDR, a single set of conversion routines is written once and is never touched again. The canonical approach has a disadvantage, but it is unimportant in real-world data transfer applications.

Suppose two little endian machines are transferring integers according to the XDR standard. The sending machine converts the integers from little-endian byte order to XDR (big-endian) byte order; the receiving machine performs the reverse conversion. Because both machines observe the same byte order, their conversions are unnecessary. The point, however, is not necessity, but cost as compared to the alternative.

The time spent converting to and from a canonical representation is insignificant, especially in networking applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure.

To transmit a tree, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there; storage for the leaf may have to be deallocated as well. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together.

Every machine pays the cost of traversing and copying data structures whether or not conversion is required. In networking applications, communications overhead—the time required to move the data down through the sender's protocol layers, across the network and up through the receiver's protocol layers—dwarfs conversion overhead.

The XDR Library

The XDR library not only solves data portability problems, it also lets you write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it can make sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

Examine the two programs more closely. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In the example, data is manipulated using standard I/O routines, so the example uses xdrstdio_create(). The parameters to XDR stream creation routines vary according to their function. In the example, xdrstdio_create() takes a pointer to an XDR structure that it initializes, a pointer to a FILE that the input or output is performed on, and the operation. The operation may be XDR_ENCODE for serializing in the writer program, or XDR_DECODE for deserializing in the reader program.

Note: RPC users never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the users.

The xdr_long Primitive

The xdr_long() primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns FALSE (0) if it fails, and TRUE (1) if it succeeds. Second, for each data type, *xxx*, there is an associated XDR routine of this form:

```
xdr_xxx(xdrs, xp)
    XDR *xdrs;
    xxx *xp;
{

}
```

In this case, *xxx* is long, and the corresponding XDR routine is a primitive, xdr_long() The client could also define an arbitrary structure *xxx*, in which case the client would also supply the routine xdr_*xxx*(), describing each field by calling XDR routines of the appropriate type. In all cases, the first parameter, xdrs can be treated as an opaque handle and passed to the primitive routines.

Direction Independence

XDR routines are direction independent; the same routines are called to serialize or deserialize data. This is critical to software engineering of portable data. The same routine is called for either operation – this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself – only in the case of deserialization is the object modified. This feature is not shown in the trivial example, but its value becomes obvious when non-trivial data structures are passed among machines. If needed, the user can obtain the direction of the XDR operation. See XDR Operation Directions for details.

Here is a slightly more complicated example. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

```
struct gnumbers
{
    long g_assets;
    long g_liabilities;
};
```

This is the corresponding XDR routine describing this structure:

```
bool_t                /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    if (xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities))
        return(TRUE);
    return(FALSE);
}
```

Note: The parameter xdrs is never inspected or modified. It is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return FALSE if the subroutine fails.

This example also shows that the type bool_t is declared as an integer whose only values are TRUE (1) and FALSE (0). This document uses these definitions:

#define bool_t	int
#define TRUE	1
#define FALSE	0

Using these conventions, xdr_gnumbers() can be rewritten in this way:

```
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    return(xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

XDR Library Primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file <xdr.h>, automatically included by <rpc.h>.

Number Filters

The XDR library provides primitives to translate between numbers and their corresponding external representations. Primitives cover the set of numbers in these formats:

```
[signed, unsigned] * [short, int, long]
```

These are the specific primitives:

```
bool_t xdr_char(xdrs, cp)
    XDR *xdrs;
    char *cp;
bool_t xdr_u_char(xdrs, ucp)
    XDR *xdrs;
    unsigned char *ucp;
bool_t xdr_int(xdrs, ip)
    XDR *xdrs;
    int *ip;
bool_t xdr_u_int(xdrs, up)
    XDR *xdrs;
    unsigned *up;
bool_t xdr_long(xdrs, lip)
    XDR *xdrs;
    long *lip;
bool_t xdr_u_long(xdrs, lup)
    XDR *xdrs;
    u_long *lup;
bool_t xdr_short(xdrs, sip)
    XDR *xdrs;
    short *sip;
bool_t xdr_u_short(xdrs, sup)
    XDR *xdrs;
    u_short *sup;
```

The first parameter, xdrs, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

Floating Point Filters

The XDR library also provides primitive routines for C floating point types, as this example shows:

```
bool_t xdr_float(xdrs, fp)
    XDR *xdrs;
    float *fp;
bool_t xdr_double(xdrs, dp)
    XDR *xdrs;
    double *dp;
```

The first parameter, xdrs is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. Both routines return TRUE if they complete successfully, and FALSE otherwise.

Note: Since the numbers are represented in IEEE floating point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice-versa.

Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C enum has the same representation inside the machine as a C integer. The boolean type is an important instance of the enum. The external representation of a boolean is always TRUE (1) or FALSE (0).

```
#define bool_t    int
#define FALSE     0
#define TRUE      1

#define enum_t    int

bool_t xdr_enum(xdrs, ep)
    XDR *xdrs;
    enum_t *ep;

bool_t xdr_bool(xdrs, bp)
    XDR *xdrs;
    bool_t *bp;
```

The second parameters ep and bp are addresses of the associated type that provides data to, or receives data from, the stream xdrs.

No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();          /* always returns TRUE     */
```

Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives previously discussed in this section. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with XDR_DECODE. Therefore, the XDR package must provide a means to deallocate memory. This is done by an XDR operation, XDR_FREE. To review, the three XDR directional operations are XDR_ENCODE, XDR_DECODE and XDR_FREE.

Strings

In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a char * and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers. Conversion between the two representations is accomplished with the routine xdr_string():

```
bool_t xdr_string(xdrs, sp, maxlength)
    XDR *xdrs;
    char **sp;
    u_int maxlength;
```

xdrs	The XDR stream handle.
sp	A pointer to a string (type char **).
maxlength	Specifies the maximum number of bytes allowed during encoding or decoding. Its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters.

Note: It is recommended that you keep maxlength small. If it is too big you can blow the heap, since xdr_string() calls malloc() for space. The routine returns FALSE if the number of characters exceeds maxlength, and TRUE if it does not.

The behavior of xdr_string() is similar to the behavior of other routines discussed in this section. The direction XDR_ENCODE is easiest to understand. The parameter sp points to a string of a certain length; if the string does not exceed maxlength, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed maxlength. Next sp is dereferenced; if the value is NULL, then a string of the appropriate length is allocated and *sp is set to this string. If the original value of *sp is non-null, then the XDR package assumes that a target area has been allocated, which can hold strings no longer than maxlength. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the XDR_FREE operation, the string is obtained by dereferencing sp. If the string is not NULL, it is freed and *sp is set to NULL. In this operation, xdr_string() ignores the maxlength parameter.

Byte Arrays

Often, variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in these ways:

- The length of the array (the byte count) is explicitly located in an unsigned integer.
- The byte sequence is not terminated by a null character.
- The external representation of the bytes is the same as their internal representation.

The primitive xdr_bytes() converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
    XDR *xdrs;
    char **bpp;
    u_int *lp;
    u_int maxlength;
```

The usage of the first, second and fourth parameters are identical to the first, second and third parameters of xdr_string(), respectively. The length of the byte area is obtained by dereferencing lp when serializing; *lp is set to the byte length when deserializing.

Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. The xdr_bytes() routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, xdr_array() requires parameters identical to those of xdr_bytes() plus two more: the size of array elements, and an XDR routine to handle each of the elements.

This routine is called to encode or decode each element of the array.

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsiz, xdr_element)
    XDR *xdrs;
    char **ap;
    u_int *lp;
    u_int maxlength;
    u_int elementsiz;
    bool_t (*xdr_element)();
```

The parameter ap is the address of the pointer to the array. If *ap is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets *ap to that array. The element count of the array is obtained from *lp when the array is serialized; *lp is set to the array length when the array is deserialized. The parameter maxlength is the maximum number of elements that the array may have; elementsiz is the byte size of each element of the array (the C function sizeof() can be used to obtain this value). The xdr_element() routine is called to serialize, deserialize, or free each element of the array.

Before defining more constructed data types, it is appropriate to present these examples.

Example A

A user on a networked machine can be identified in these ways:

- By the machine name, such as krypton: see the gethostname man page
- By the user's UID
- By the group numbers to which the user belongs: see the getgroups man page

A structure with this information and its associated XDR routine could be coded like this:

```
struct netuser
{
    char     *nu_machinename;
    int      nu_uid;
    u_int    nu_glen;
    int      *nu_gids;
};
#define NLEN 255               /* machine names < 256 chars */
#define NGRPS 20               /* user can't be in > 20 groups */

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    return(xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
    xdr_int(xdrs, &nup->nu_uid) &&
    xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen,
    NGRPS, sizeof (int), xdr_int));
}
```

Example B

A group of network users could be implemented as an array of netuser structure. This is the declaration and its associated XDR routines:

```
struct party
{
    u_int p_len;
    struct netuser *p_nusers;
};
#define PLEN 500          /* max number of users in a party */

bool_t
xdr_party(xdrs, pp)
    XDR *xdrs;
    struct party *pp;
{
    return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
        sizeof (struct netuser), xdr_netuser));
}
```

Example C

The well-known parameters to main, argc, and argv can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like this:

```
struct cmd
{
    u_int     c_rgc;
    char      **c_argv;
};
#define ALEN 1000            /* args cannot be > 1000 chars */
#define NARGC 100            /* commands cannot have > 100 args */

struct history
{
    u_int          h_len;
    struct cmd     *h_cmds;
};
#define NCMDS 75                    /* history is no more than 75 commands */

bool_t
xdr_wrap_string(xdrs, sp)
    XDR     *xdrs;
    char    **sp;
{
    return(xdr_string(xdrs, sp, ALEN));
}
bool_t
xdr_cmd(xdrs, cp)
    XDR            *xdrs;
    struct cmd     *cp;
{
    return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
           sizeof (char *), xdr_wrap_string));
}
bool_t
xdr_history(xdrs, hp)
    XDR              *xdrs;
    struct history   *hp; ]
{ ]    return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
           sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine xdr_wrap_string() is needed to package the xdr_string() routine, because the implementation of xdr_array() only passes two parameters to the array element description routine; xdr_wrap_string() supplies the third parameter to xdr_string().

By now the recursive nature of the XDR library should be obvious. The following examples continue with more constructed data types.

Opaque Data

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The xdr_opaque() primitive is used for describing fixed sized, opaque bytes.

```
bool_t xdr_opaque(xdrs, p, len)
    XDR     *xdrs;
    char    *p;
    u_int    len;
```

The parameter p is the location of the bytes; len is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object is not machine portable.

Fixed Sized Arrays

The XDR library provides a primitive, xdr_vector(), for fixed-length arrays.

```
#define NLEN 255          /* machine names must be < 256 chars */
#define NGRPS 20          /* user belongs to exactly 20 groups */

struct netuser
{
    char    *nu_machinename;
    int     nu_uid;
    int     nu_gids[NGRPS];
};

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    int i;

    if (!xdr_string(xdrs, &nup->nu_machinename, NLEN))
        return(FALSE);
    if (!xdr_int(xdrs, &nup->nu_uid))
        return(FALSE);
    if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int), xdr_int))
        {
            return(FALSE);
        }
    return(TRUE);
}
```

Discriminated Unions

The XDR library supports discriminated unions. A discriminated union is a C union and an enum_t value that selects an "arm" of the union.

```
struct xdr_discrim
{
    enum_t     value;
    bool_t     (*proc)();
};
bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
    XDR                    *xdrs;
    enum_t                 *dscmp;
    char                   *unp;
    struct xdr_discrim     *arms;
    bool_t                 (*defaultarm)();   /* may equal NULL */
```

The routine translates the discriminant of the union located at *dscmp. The discriminant is always an enum_t. The union located at *unp is then translated. The parameter arms is a pointer to an array of xdr_discrim structures.

Each structure contains an ordered pair of [value,proc]. If the union's discriminant is equal to the associated value, then the proc is called to translate the union. The end of the xdr_discrim structure array is denoted by a routine of value NULL (0). If the discriminant is not found in the arms array, then the defaultarm procedure is called if it is non-null; otherwise the routine returns FALSE.

Example D

Suppose the type of a union may be integer, character pointer (a string), or a gnumbers structure. Also, assume the union and its current type are declared in a structure. This is the declaration:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };
struct u_tag
{
    enum utype utype;          /* the union's discriminant */
    union
    {
        int                ival;
        char               *pval;
        struct gnumbers    gn;
    }
    uval;
};
```

These constructs and XDR procedure (de)serialize the discriminated union.

```
struct xdr_discrim u_tag_arms[4] =
{
    { INTEGER, xdr_int },
    { GNUMBERS, xdr_gnumbers }
    { STRING, xdr_wrap_string },
    { __dontcare__, NULL }

    /* always terminate arms with a NULL xdr_proc */
}

bool_t
xdr_u_tag(xdrs, utp)
    XDR             *xdrs;
    struct u_tag    *utp;
{
    return(xdr_union(xdrs, &utp->utype, &utp->uval,
                        u_tag_arms, NULL));
}
```

The routine xdr_gnumbers() was presented in The XDR Library . xdr_wrap_string() was presented in Example C. The default arm parameter to xdr_union() (the last parameter) is NULL in this example. Therefore the value of the union's discriminant may legally take on only values listed in the u_tag_arms array. This example also demonstrates that the elements of the arms array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Pointers

In C, it is often convenient to put pointers to another structure within a structure. The xdr_reference() primitive makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t xdr_reference(xdrs, pp, size, proc)
    XDR         *xdrs;
    char        **pp;
    u_int         ssize;
    bool_t        (*proc)();
```

Parameter pp is the address of the pointer to the structure; parameter ssize is the size in bytes of the structure (use the C function sizeof() to obtain this value); and proc is the XDR routine that describes the structure.

When decoding data, storage is allocated if *pp is NULL. There is no need for a primitive xdr_struct() to describe structures within structures, because pointers are always sufficient.

Example E

Suppose there is a structure containing a person's name and a pointer to a gnumbers structure containing the person's gross assets and liabilities.

This is the construct:

```
struct pgn
{
    char *name;
    struct gnumbers *gnp;
};
```

This is the corresponding XDR routine for this structure:

```
bool_t
xdr_pgn(xdrs, pp)
    XDR *xdrs;
    struct pgn *pp;
{
    if (xdr_string(xdrs, &pp->name, NLEN) &&
        xdr_reference(xdrs, &pp->gnp,
        sizeof(struct gnumbers), xdr_gnumbers))
            return(TRUE);
        return(FALSE);
}
```

Pointer Semantics and XDR

In many applications, C programmers attach double meaning to the values of a pointer. Typically, the value NULL (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer.

In Example E, a NULL pointer value for gnp could indicate that the person's assets and liabilities are unknown). The pointer value encodes two pieces of information: whether or not the data is known; and if it is known, where it is located in memory.

Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive xdr_reference() cannot and does not attach any special meaning to a null-value pointer during serialization – passing an address of a pointer whose value is NULL to xdr_reference() when serializing data most likely causes a memory fault and, on the UNIX system, a core dump. xdr_pointer() correctly handles NULL pointers. For more information about its use, see Advanced Topics .

Non-Filter Primitives

XDR streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
    XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
    XDR *xdrs;
    u_int pos;

xdr_destroy(xdrs)
    XDR *xdrs;
```

The routine xdr_getpos() returns an unsigned integer that describes the current position in the data stream.

In some XDR streams, the returned value of xdr_getpos() is meaningless; the routine returns a -1 in this case (though -1 should be a legitimate value).

The routine xdr_setpos() sets a stream position to positive.

In some XDR streams, setting a position is impossible; in such cases, xdr_setpos() returns FALSE. This routine also fails if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The xdr_destroy() primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

XDR Operation Directions

At times you may want to optimize XDR routines by taking advantage of the direction of the operation – XDR_ENCODE, XDR_DECODE or XDR_FREE. The value xdrs->x_op always contains the direction of the XDR operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in Memory Streams demonstrates the usefulness of the xdrs->x_op field.

XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream. Streams currently exist for (de)serialization of data to or from standard I/O FILE streams, TCP/IP connections, and memory.

Standard I/O Streams

XDR streams can be interfaced to standard I/O using this xdrstdio_create() routine:

```
#include <stdio.h>
#include <rpc.h>               /* xdr streams part of rpc */

void
xdrstdio_create(xdrs, fp, x_op)
    XDR *xdrs;
    FILE *fp;
    enum xdr_op x_op;
```

The routine xdrstdio_create() initializes an XDR stream pointed to by xdrs. The XDR stream interfaces to the standard I/O library. Parameter fp is an open file, and x_op is an XDR direction.

Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory, as shown below:

```
#include <rpc.h>

void
xdrmem_create(xdrs, addr, len, x_op)
    XDR           *xdrs;
    char          *addr;
    u_int         len;
    enum xdr_op     x_op;
```

The routine xdrmem_create() initializes an XDR stream in local memory. The memory is pointed to by parameter addr; parameter len is the length in bytes of the memory. The parameters xdrs and x_op are identical to the corresponding parameters of xdrstdio_create(). Currently, the UDP/IP implementation of RPC uses xdrmem_create(). Complete call or result messages are built in memory before calling the sendto() system routine.

Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the TCP socket.

```
#include <rpc.h>               /* xdr streams part of rpc */

xdrrec_create(xdrs,
    sendsize, recvsize, iohandle, readproc, writeproc)
        XDR        *xdrs;
        u_int       sendsize, recvsize;
        char       *iohandle;
        int         (*readproc)(), (*writeproc)();
```

The routine xdrrec_create() provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter xdrs is similar to the corresponding parameter previously described. The stream does its own data buffering similar to that of standard I/O. The parameters sendsize and recvsize determine the size in bytes of the output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used.

When a buffer needs to be filled or flushed, the routine readproc() or writeproc() is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls read() and write(). However, the first parameter to each of these routines is the opaque parameter iohandle. The other two parameters (buf and nbytes) and the results (byte count) are identical to the system routines. If xxx is readproc() or writeproc(), then it has these form:

```
/* returns the actual number of bytes transferred
 * -1 is an error */

int
xxx(iohandle, buf, len)
    char *iohandle;
    char *buf;
    int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in Advanced Topics. The primitives that are specific to record streams are:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
    XDR   *xdrs;
    bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
    XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
    XDR *xdrs;
```

The routine xdrrec_endofrecord() causes the current outgoing data to be marked as a record. If the flushnow parameter is TRUE, then the stream's writeproc is called; otherwise, writeproc is called when the output buffer has been filled.

The routine xdrrec_skiprecord() causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream. If there is no more data in the stream's input buffer, then the routine xdrrec_eof() returns TRUE. That is not to say that there is no more data in the underlying file descriptor.

XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

The XDR Object

This structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2 };
typedef struct
{
    enum xdr_op x_op;         /* operation; fast added param */
    struct xdr_ops
    {
        bool_t  (*x_getlong)();     /* get long from stream */
        bool_t  (*x_putlong)();     /* put long to stream */
        bool_t  (*x_getbytes)();    /* get bytes from stream */
        bool_t  (*x_putbytes)();    /* put bytes to stream */
        u_int   (*x_getpostn)();    /* return stream offset */
        bool_t  (*x_setpostn)();    /* reposition offset */
        caddr_t (*x_inline)();      /* ptr to buffered data */
        VOID    (*x_destroy)();     /* free private area */
    }
    *x_ops;
        caddr_t x_public;           /* users' data */
        caddr_t x_private;          /* pointer to private data */
        caddr_t x_base;             /* private for position info */
        int x_handy;                /* extra private word */
}
XDR;
```

The x_op field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields x_private, x_base, and x_handy are private to the particular stream's implementation. The field x_public is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives. x_getpostn() x_setpostn(), and x_destroy() are macros for accessing operations. The operation x_inline() takes two parameters: an XDR *, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose.

From the stream's point of view, the bytes in the buffer segment have been consumed or put. The routine may return NULL if it cannot return a buffer segment of the requested size.

Note: The x_inline()routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.

The operations x_getbytes() and x_putbytes() blindly get and put sequences of bytes from or to the underlying stream; they return TRUE if they are successful, and FALSE otherwise. The routines have identical parameters (replace xxx):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
    XDR *xdrs;
    char *buf;
    u_int bytecount;
```

The operations x_getlong() and x_putlong() receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The network utility primitives htonl() and ntohl() can be helpful in accomplishing this.

The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return TRUE if they succeed, and FALSE otherwise. They have identical parameters. Here is an example:

```
bool_t
xxxlong(xdrs, lp)
    XDR  *xdrs;
    long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the examples in this section are written using both the XDR C library routines and the XDR data description language. Read the XDR Protocol Specification, RFC 1014, for a complete description of this language.

Linked Lists

The previous example in Example D presented a C data structure and its associated XDR routines for an individual's gross assets and liabilities. This example is duplicated here:

```
struct gnumbers
{
    long g_assets;
    long g_liabilities;
};
bool_t
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    if (xdr_long(xdrs, &(gp->g_assets)))
        return(xdr_long(xdrs, &(gp->g_liabilities))); return(FALSE);
}
```

To implement a linked list of such information, a data structure could be constructed like this:

```
struct gnumbers_node
{
    struct gnumbers gn_numbers;
    struct gnumbers_node *gn_next;
};

typedef struct gnumbers_node *gnumbers_list;
```

The head of the linked list can be thought of as the data object (i.e., the head is not merely a convenient shorthand for a structure). Similarly the gn_next field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the gn_next field is also the address of where it continues.

Serialized Objects

The link addresses carry no useful information when the object is serialized. The XDR data description of this linked list is described by this recursive declaration of gnumbers_list:

```
struct gnumbers
{
    int g_assets;
    int g_liabilities;
};

struct gnumbers_node
{
    gnumbers gn_numbers;
    gnumbers_node *gn_next;
};
```

In this description, the boolean indicates whether there is more data following it. If the boolean is FALSE, then it is the last data field of the structure. If it is TRUE, then it is followed by a gnumbers structure and (recursively) by a gnumbers_list.

Note: The C declaration has no boolean explicitly declared in it (though the gn_next field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for Writing XDR Routines

Hints for writing the XDR routines for a gnumbers_list follow easily from the previous XDR description. The primitive xdr_pointer() is used to implement the previous XDR union:

```
bool_t
xdr_gnumbers_node(xdrs, gn)
    XDR *xdrs;
    gnumbers_node *gn;
{
    return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
        xdr_gnumbers_list(xdrs, &gp->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    gnumbers_list *gnp;
{
    return(xdr_pointer(xdrs, gnp,
        sizeof(struct gnumbers_node),
        xdr_gnumbers_node));
}
```

A Non-Recursive Example

The unfortunate side effect of XDRing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. This routine collapses the previous two mutually recursive procedures into a single, non-recursive one.

```
bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    numbers_list *gnp;
{
    bool_t more_data;
    gnumbers_list *nextp;

    for (;;)
    {
        more_data = (*gnp != NULL);
        if (!xdr_bool(xdrs, &more_data))
        {
            return(FALSE);
        }
        if (! more_data)
        {
            break;
        }
        if (xdrs->x_op == XDR_FREE)
        {
            nextp = &(*gnp)->gn_next;
        }
        if (!xdr_reference(xdrs, gnp, sizeof(struct
                          gnumbers_node), xdr_gnumbers))
            {
                return(FALSE);
            }
    gnp = (xdrs->x_op == XDR_FREE) ?
        nextp : &(*gnp)->gn_next;
    }
    *gnp = NULL;
    return(TRUE);
}
```

Tasks Performed

This example performs these tasks:

1. Finds out whether there is more data or not, so that this boolean information can be serialized.

   This statement is unnecessary in the XDR_DECODE case, since the value of more_data is not known until it is deserialized in the next statement.

2. Does an XDR on the more_data field of the XDR union. Then if there is truly no more data, the last pointer is set to NULL to indicate the end of the list, and returns TRUE because it is done.

   Setting the pointer to NULL is only important in the XDR_DECODE case, since it is already NULL in the XDR_ENCODE and XDR_FREE cases.

3. If the direction is XDR_FREE, the value of nextp is set to indicate the location of the next pointer in the list. This is done now because gnp needs to be de-referenced to find the location of the next item in the list, and after the next statement the storage pointed to by gnp is freed up and is no longer valid. This cannot be done for all directions though, because in the XDR_DECODE direction, the value of gnp is not set until the next statement.

4. An XDR is done on the data in the node using the primitive xdr_reference(). xdr_reference() is like xdr_pointer(), which was previously used, but it does not send over the boolean indicating whether there is more data. It is used instead of xdr_pointer() because you already did an XDR on this information.

   The XDR routine passed is not the same type as an element in the list. The routine passed is xdr_gnumbers(), for XDRing gnumbers, but each element in the list is actually of type gnumbers_node. xdr_gnumbers_node() is not passed because it is recursive; xdr_gnumbers() is used instead, which does an XDR on all of the non-recursive part.

   This works only if the gn_numbers field is the first item in each element, so that their addresses are identical when passed to xdr_reference().

5. gnp is updated to point to the next item in the list. If the direction is XDR_FREE, it is set to the previously saved value, otherwise gnp is de-referenced to get the proper value.

   Though harder to understand than the recursive version, this non-recursive routine is far less likely to blow the C stack. It also runs more efficiently since a lot of procedure call overhead has been removed. Most lists are small, though, (in the hundreds of items or less) and the recursive version should be sufficient for them.

Chapter 4

Using rpcgen

This chapter describes the rpcgen compiler. It includes these sections:

- What rpcgen Does—Describes the rpcgen compiler and its input, output, and interfaces
- Converting Local Procedures into Remote Procedures—Uses a printmessage example to illustrate converting a local procedure to a remote procedure. Describes the RPC steps involved and the steps for completing the conversion process
- Generating XDR Routines—Provides an example protocol description file and explains XDR routines for converting data types as well as how to test the client and server procedures together
- The C Preprocessor—Describes the symbols that may be defined and includes a description of rpcgen preprocessing
- rpcgen Programming Notes—Includes timeout changes, handling broadcast on the server side, and other information passed to server procedures
- The RPC Language—Describes definitions, structures, unions, enumerations, typedefs, constants, programs, declarations, and special cases

What rpcgen Does

The rpcgen compiler exists to help you write RPC applications simply and directly. rpcgen does most of the work, letting you debug the main features of your application, instead of requiring you to spend most of your time debugging your network interface code.

Note: The rpcgen compiler is not supplied with the Unicenter TCPaccess Communications Server RPC/XDR product but may be available on remote workstations. It can be useful to generate C language output, which can then be transferred to the mainframe.

How rpcgen Works

rpcgen is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output that includes stub versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, and a header file that contains common definitions.

The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that are invoked by remote clients. rpcgen's output files can be compiled and linked in the usual way.

You write server procedures and link them with the server skeleton, produced by rpcgen, to get an executable server program. To use a remote program, you write an ordinary main program that makes local procedure calls to the client stubs produced by rpcgen. Linking this program with rpcgen's stubs creates an executable program. (At present, the main program must be written in C). rpcgen options can be used to suppress stub generation and to specify the transport to be used by the server stub.

Like all compilers, rpcgen reduces development time that would otherwise be spent coding and debugging low-level routines at a small cost in efficiency and flexibility. In addition, like many compilers, rpcgen allows escape hatches for programmers to mix low-level code with high-level code. In speed-critical applications, hand-written routines can be linked with the rpcgen output without any difficulty. Also, you may proceed by using rpcgen output as a starting point, and then rewriting it as necessary. For a discussion of RPC programming without rpcgen, see Using Remote Procedure Calls.

Converting Local Procedures into Remote Procedures

One task that may need to be done is to convert an application that runs on a single machine to one that runs over the network.

A printmessage Example

The following example—a program, printmessage, that prints a message to the console—is converted so that a message can be sent to the console from anywhere in the system:

```
/* printmsg.c: print a message on the console */

#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    char *message;

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s <message>\n", argv[0]);
        exit(1);
    }
    message = argv[1];

    if (!printmessage(message))
    {
        fprintf(stderr, "%s: couldn't print your message\n", argv[0]);
        exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}

    /* Print a message to the console. Return a boolean indicating
     * whether the message was actually printed. */

printmessage(msg)
    char *msg;
{
    FILE *f;

    f = fopen("/dev/console", "w");
    if (f == NULL)
    {
        return (0);
    }
    fprintf(f, "%s\n", msg);
    fclose(f);
    return(1);
}
And then, of course:
example% cc printmsg.c -o printmsg
example% printmsg "Hello, there."
Message delivered!
example%
```

Remote Procedures Steps

If printmessage is turned into a remote procedure, it can be called from anywhere in the network. Ideally, you would only insert a keyword, like remote, in front of a procedure to turn it into a remote procedure.

Determine Procedure Input and Output Types

You must first determine what types there are for all procedure inputs and outputs. In this example, printmessage takes a string as input, and returns an integer as output. Knowing this, you write a protocol specification like this in RPC language that describes the remote version of printmessage:

```
/* msg.x: Remote message printing protocol */
program MESSAGEPROG
{
    version MESSAGEVERS
    {
        int PRINTMESSAGE(string) = 1;
    } = 1;
} = 99;
```

Remote procedures are part of remote programs, so an entire remote program is actually declared here that contains the single procedure PRINTMESSAGE. This procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary because rpcgen generates it automatically.

In this example, the argument type is string and not char *. This is because a char * in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a string.

The Remote Procedure

You next write the actual remote procedure. The following is the definition of a remote procedure to implement the PRINTMESSAGE procedure previously declared:

```
/* msg_proc.c: implementation of the remote procedure "printmessage" */
#include <stdio.h>
#include <rpc.h>           /* always needed */
#include "msg.h"           /* msg.h will be generated by rpcgen */

/* Remote version of "printmessage" */

int *
printmessage_1(msg)
    char **msg;{
    static int result;         /* must be static! */
    FILE *f;
    f = fopen("/dev/console", "w");
    if (f == NULL)
    {        result = 0;
        return (&result);
    }    fprintf(f, "%s\n", *msg);
    fclose(f);
    result = 1;
    return (&result);
}
```

The declaration of the remote procedure printmessage_1 differs from that of the local procedure printmessage in these ways:

- It takes a pointer to a string instead of a string itself. This is true of all remote procedures: they always take pointers to their arguments rather than the arguments themselves.
- It returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures: they always return a pointer to their results.
- It has a _1 appended to its name. In general, all remote procedures called by rpcgen are named by this rule: the name in the program definition (here PRINTMESSAGE) is converted to all lower-case letters, an underscore (_) is appended to it, and the version number (here 1) is appended.

Declare the Main Client Program

The last step is to declare the main client program that calls the remote procedure. The following is an example:

```
/* rprintmsg.c: remote version of "printmsg.c" */
#include <stdio.h>
#include <rpc.h>           /* always needed */
#include "msg.h"            /* msg.h will be generated by rpcgen */

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    int *result;
    char *server;
    char *message;

    if (argc != 3)
    {
        fprintf(stderr,
        "usage: %s host message\n", argv[0]);
        exit(1);
    }

    /* Save values of command line arguments */

    server = argv[1];
    message = argv[2];

    /* Create client "handle" used for calling MESSAGEPROG on the
     * server designated on the command line. We tell the RPC
     * package to use the "tcp" protocol when contacting
     * the server. */

    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
        if (cl == NULL)
        {

            /* *Couldn't establish connection with server.
             * Print error message and die. */

            clnt_pcreateerror (server);
            exit(1);
        }
    /* Call the remote procedure "printmessage" on the server */

    result = printmessage_1(&message, cl);
    if (result == NULL)
    {
        /* An error  occurred while calling the server.
         * Print error message and die. */

        clnt_perror (cl, server);
        exit(1);
    }

    /* Okay, we successfully called the remote procedure. */

    if (*result == 0)
    {

        /* Server was unable to print our message.
         * Print error message and die. */
```

```
        fprintf(stderr, "%s: %s couldn't print your message\n",
                        argv[0], server);
            exit(1);
    }

    /* The message got printed on the server's console */

    printf("Message delivered to %s!\n", server);
    exit(0);
}
```

The client handle (called handle in the example) used by rpcgen is created using the RPC library routine clnt_create. This client handle is passed to the stub routines that call the remote procedure.

The remote procedure printmessage_1 is called exactly the same way as it is declared in msg_proc.c except for the inserted client handle as the first argument.

Completing the Process

The following example shows how to complete the process:

```
example% rpcgen msg.x
example% cc rprintmsg.c msg_clnt.c -o rprintmsg
example% cc msg_proc.c msg_svc.c -o msg_server
```

Two programs were compiled:

- The client program rprintmsg
- The server program msg_server.

Before doing this, rpcgen was used to fill in the missing pieces.

rpcgen did the following with the input file msg.x:

- It created a header file called msg.h that contained #defines for MESSAGEPROG, MESSAGEVERS and PRINTMESSAGE for use in the other modules.
- It created client stub routines in the msg_clnt.c file. In this case, there is only one, the printmessage_1 referred to from the printmsg client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is FOO.x, the client stubs output file is called FOO_clnt.c.
- It created the server program that calls printmessage_1 in msg_proc.c. This server program is named msg_svc.c. The rule for naming the server output file is similar to the previous one: for an input file called FOO.x, the output server file is named FOO_svc.c.

You are now ready to test the example. First, copy the server to a remote machine and run it. In this example, the machine is called moon.

```
moon% msg_server &
```

Server processes are run in the background, because they never exit. Then, on the local machine sun, print a message on moon's console.

```
sun% rprintmsg moon "Hello, moon."
```

The message gets printed to moon's console. You can print a message on anybody's console (including your own) with this program if you are able to copy the server to their machine and run it.

Generating XDR Routines

The previous example only demonstrated the automatic generation of client and server RPC code. rpcgen may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice-versa.

This example presents a complete RPC service – a remote directory listing service, which uses rpcgen not only to generate stub routines, but also to generate the XDR routines.

Protocol Description File

The following is the protocol description file:

```
/* dir.x: Remote directory listing protocol */

const MAXNAMELEN = 255;   /* maximum length of a directory entry */
typedef string nametype<MAXNAMELEN>;    /* a directory entry */
typedef struct namenode *namelist;      /* a link in the listing */

/* A node in the directory listing */

struct namenode
{
    nametype name         /* name of directory entry */
    namelist next;        /* next entry   */
};
                          /* The result of a READDIR operation. */
union readdir_res switch (int errno)
{
    case 0:
        namelist list;  /* no error : return directory listing */
    default:
        void;           /* error  occurred: nothing else to return */
};

/* The directory program definition */
```

```
program DIRPROG
{
    version DIRVERS
    {
        readdir_res
        READDIR(nametype) = 1;
    } = 1;
} = 76;
```

Types (like readdir_res in this example) can be defined using the struct, union, and enum keywords, but those keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union foo, you should declare using only foo and not union foo. rpcgen compiles RPC unions into C structures and it is an error to declare them using the union keyword.

XDR Routines for Converting Data Types

Running rpcgen on dir.x creates four output files. Three are the same as before: header file, client stub routines, and server skeleton. The fourth file contains the XDR routines necessary for converting the data types declared into XDR format and vice-versa. These are output in the file dir_xdr.c.

The READDIR Procedure

The following is the implementation of the READDIR procedure:

```
/* dir_proc.c: remote readdir implementation */

#include <rpc.h>
#include <dir.h>
#include "dir.h"

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(dirname)
    nametype *dirname;
{
    DIR *dirp;
    struct direct *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res;        /* must be static! */

    /* Open directory */

    dirp = opendir(*dirname);
    if (dirp == NULL)
    {
        res.errno = errno;
        return (&res);
```

The Client-Side Program to Call the Server

The following is the client-side program to call the server:

```
/* rls.c: Remote directory listing client */
#include <stdio.h>
#include <rpc.h>              /* always need this */
#include "dir.h"              /* will be generated by rpcgen */

extern int errno;

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;

    if (argc != 3)
    {
        fprintf(stderr, "usage: %s host directory\n", argv[0]);
        exit(1);
    }

    /* Remember what our command line arguments refer to */

    server = argv[1];
    dir = argv[2];

    /* Create client "handle" used for calling MESSAGEPROG on the
     * server designated on the command line. We tell the RPC
     * package to use the "tcp" protocol when contacting
     * the server. */

    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
    if (cl == NULL)
    {

        /* Couldn't establish connection with server.
         * Print error  message and die. */

        clnt_pcreateerror (server);
        exit(1);
    }

    /* Call the remote procedure readdir on the server */

    result = readdir_1(&dir, cl);
    if (result == NULL)
    {

        /* An error  occurred while calling the server.
         * Print error  message and die. */

        clnt_perror (cl, server);
        exit(1);
    }
    /* Okay, we successfully called the remote procedure. */

    if (result->errno != 0)
    {
```

```
/* A remote system error  occurred. Print error
         * message and die. */
        errno = result->errno;
        perror(dir);
        exit(1);
    }

    /* Successfully got a directory listing. Print it out. */

    for (nl = result->readdir_res_u.list;
        nl != NULL;
        nl = nl->next)
        {
            printf("%s\n", nl->name);
        }
    exit(0);
}
```

Compiling and Running

Compile everything, and then run the following routine:

```
sun% rpcgen dir.x
sun% cc rls.c dir_clnt.c dir_xdr.c -o rls
sun% cc dir_svc.c dir_proc.c dir_xdr.c -o dir_svc

sun% dir_svc &

moon% rls sun /usr/pub
.
.
.
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
moon%
```

Testing the Client and Server Procedures Together

A final note about rpcgen. The client program and the server procedure can be tested together as a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls are executed as ordinary local procedure calls and the program can be debugged with a local debugger. When the program is working, the client program can be linked to the client stub produced by rpcgen and the server procedures can be linked to the server stub produced by rpcgen.

If you do this, you may want to comment out calls to RPC library routines, and have client-side routines call server routines directly.

The C Preprocessor

The C preprocessor is run on all input files before they are compiled, so all the preprocessor directives are legal within an .x file.

Symbols That May Be Defined

The following symbols may be defined, depending on which output file is being generated:

Symbol	Description
RPC_HDR	For header-file output.
RPC_XDR	For XDR routine output.
RPC_SVC	For server-skeleton output.
RPC_CLNT	For client stub output.

rpcgen Preprocessing

rpcgen does some preprocessing of its own. Any line that begins with a percent sign is passed directly into the output file without any interpretation of the line. This example demonstrates the preprocessing features:

```
/* time.x: Remote time protocol */

program TIMEPROG
{
    version TIMEVERS
    {
        unsigned int TIMEGET(void) = 1;
    } = 1;
} = 44;
#ifdef RPC_SVC
%int *
%timeget_1()
%{
%     static int thetime;
%
%     thetime = time(0);
%     return (&thetime);
%}
#endif
```

The % feature is not generally recommended, as there is no guarantee that the compiler puts the output where you intended.

rpcgen Programming Notes

This section contains useful notes on rpcgen programming.

Timeout Changes

RPC sets a default timeout of 25 seconds for RPC calls when clnt_create is used. This timeout may be changed using clnt_control.

Here is a small code fragment to demonstrate use of clnt_control:

```
struct timeval tv;
CLIENT *cl;

cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");
if (cl == NULL)
{
    exit(1);
}
tv.tv_sec = 60;                 /* change timeout to 1 minute */
tv.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

Handling Broadcast on the Server Side

When a procedure is known to be called via broadcast RPC, the server should reply only if it can provide some useful information to the client.

To prevent the server from replying, a remote procedure can return NULL as its result, and the server code generated by rpcgen detects this and does not send out a reply.

The following is an example of a procedure that replies only if it thinks it is an NFS server:

```
void *
reply_if_nfsserver()
{
    char notnull;        /* just here so we can use its address */

    if (access("/etc/exports", F_OK) < 0)
    {
        return (NULL);           /* prevent RPC from replying */
    }

    /* Return non-null pointer so RPC will send out a reply */

    return ((void *)&notnull);
}
```

For example, if the procedure returns type void *, if must return a non-null pointer if wants RPC to reply for it.

Other Information Passed to Server Procedures

Server procedures often want to know more about an RPC call than just its arguments.

Getting authentication information is important to procedures that want to implement some level of security.

This extra information is actually supplied to the server procedure as a second argument.

The following is an example to demonstrate its use. Rewrite the previous printmessage_1 procedure to only let root users print a message to the console:

```
int *
%
printmessage_1(msg, rq)
    char **msg;
    struct svc_req *rq;
{
    static in result;              /* Must be static */
    FILE *f;
    struct suthunix_parms *aup;

    aup = (struct authunix_parms *)rq->rq_clntcred;
    if (aup->aup_uid != 0)
    {
        result = 0;
        return (&result);
    }
    .
    .  Same code as before
    .
}
```

The RPC Language

The RPC language is an extension of XDR language. The sole extension is the addition of the program type. For a complete description of the XDR language syntax, read the XDR Protocol specification, RFC 1014. For a description of the RPC extensions to the XDR language, read RFC 1057.

The XDR language is very close to C; you know C, you know most of XDR. The following topics describe the syntax of the RPC language, showing a few examples, and showing how the various RPC and XDR type definitions get compiled into C type definitions in the output header file.

Definitions

An RPC language file consists of a series of definitions:

```
definition-list:
    definition ";"
    definition ";" definition-list
```

It recognizes these types of definitions:

```
definition:
    enum-definition
    struct-definition
    union-definition
    typedef-definition
    const-definition
    program-definition
```

Structures

An XDR struct is declared almost exactly like its C counterpart:

```
struct-definition:

    "struct" struct-ident "{"

        declaration-list

    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

The following shows an XDR structure to a two-dimensional coordinate, and the C structure that gets compiled into the output header file:

```
struct coord {
                int x;
                int y;
            };
becomes
struct coord {
                int x;
                int y;
            };
            typedef struct coord coord;
```

The output is identical to the input, except for the added typedef at the end of the output. This lets you use coord instead of struct coord when declaring items.

Unions

XDR unions are discriminated unions, and look quite different from C unions. They are more analogous to Pascal variant records than they are to C unions:

```
union-definition:
    "union" union-ident "switch" "(" declaration ")" "{"
        case-list
    "}"

case-list:
    "case" value ":" declaration ";"
    "default" ":" declaration ";"
    "case" value ":" declaration ";" case-list
```

The following is an example of a type that might be returned as the result of a read data operation. If there is no error, return a block of data. Otherwise, do not return anything:

```
union read_result switch (int errno)
{
    case 0:
        opaque data[1024];
    default:
        void;
};
```

... gets compiled into this code:

```
struct read_result
{
    int errno;
    union
    {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

The union component of the output struct has the same name as the type, except for the trailing _u.

Enumerations

XDR enumerations have the same syntax as C enumerations:

```
enum-definition:
    "enum" enum-ident "{"
    enum-value-list
    "}"

enum-value-list:
    enum-value
    enum-value "," enum-value-list

enum-value:
    enum-value-ident
    enum-value-ident "=" value

The following a short example of an XDR enum, and the C enum that it gets compiled
into:
enum colortype {
                  RED = 0,
                  GREEN = 1,
                  BLUE = 2
                  };
...gets compiled into:
enum colortype {
                  RED = 0,
                  GREEN = 1,
                  BLUE = 2
                  };
typedef enum colortype colortype;
```

Typedefs

XDR typedefs have the same syntax as C typedefs:

```
typedef-definition:
    "typedef" declaration
```

The following is an example that defines a fname_type used for declaring file name strings that have a maximum length of 255 characters:

```
typedef string fname_type<255>;
```

becomes

```
typedef char *fname_type;
```

Constants

XDR contains symbolic constants that may be used wherever an integer constant is used; for example, in array size specifications:

```
const-definition:
    "const" const-ident "=" integer
```

The following example shows how to define the constant DOZEN equal to 12:

```
const DOZEN = 12;
```

becomes

```
#define DOZEN 12
```

Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value
version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value

procedure-list:
    procedure ";"
        procedure ";" procedure-list

procedure:
    type-ident procedure-ident "(" type-ident ")" "=" value
```

Here is the time protocol, revisited:

```
/* time.x: Get or set the time. Time is represented
* as number of seconds since 0:00, January 1, 1970. */

program TIMEPROG
{
    version TIMEVERS
    {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 44;
```

This file compiles into #defines in the output header file:

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Declarations

XDR supports the following types of declarations:

- Simple-declaration
- Fixed-array-declaration
- Variable-array-declaration
- Pointer-declaration

Simple Declarations

These are just like simple C declarations:

```
simple-declaration:
    type-ident variable-ident
```

For example,

```
colortype color;
```

becomes

```
colortype color;
```

Fixed-length Array Declarations

These are just like C array declarations:

```
fixed-array-declaration:
    type-ident variable-ident "[" value "]"
```

For example,

```
colortype palette[8];
```

becomes

```
colortype palette[8];
```

Variable-Length Array Declarations

Variable-Length array declarations have no explicit syntax in C. The XDR format uses angle brackets:

```
variable-array-declaration:
    type-ident variable-ident "<" value ">"
    type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets. The size may be omitted, which indicates that the array may be of any size:

```
int heights<12>;                /* at most 12 items */
int widths<>;                   /* any number of items */
```

Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into structs.

For example, the heights declaration gets compiled into the following struct:

```
struct
{
    u_int heights_len;              /* # of items in array */
    int *heights_val;               /* pointer to array */
    } heights;
```

The number of items in the array is stored in the _len component and the pointer to the array is stored in the _val component. The first part of each component name is the same as the name of the declared XDR variable.

Pointer Declarations

Pointer declarations are made in XDR exactly as they are in C. You cannot send pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees. The type is actually called “optional-data” in XDR language, not “pointer”:

```
pointer-declaration:
    type-ident "*" variable-ident
```

For example,

```
listitem *next; --> listitem *next;
```

Special Cases

There are a few exceptions to the previously described rules.

Booleans

C has no built-in boolean type. However, the RPC library does have a boolean type called bool_t that is either TRUE or FALSE. Things declared as type bool in XDR language are compiled into bool_t in the output header file.

```
bool married;
```

becomes

```
bool_t married;
```

Strings

C has no built-in string type, but instead uses the null-terminated char * convention. In XDR language, strings are declared using the string keyword and compiled into char *s in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the NULL character). The maximum size may be left off, which indicates a string of arbitrary length.

The following are two examples:

```
string name<32>;
```

becomes

```
char *name;
```

The second example:

```
string longname<>;
```

becomes

```
char *longname;
```

Opaque Data

Opaque data is used in RPC and XDR to describe untyped data, that is, just sequences of arbitrary bytes. It may be declared either as a fixed or variable length array.

The following are two examples:

```
opaque diskblock[512];
```

becomes

```
char diskblock[512];
```

The second example is:

```
opaque filedata<1024>;
```

becomes

```
struct
{
    u_int filedata_len;
    char *filedata_val;
} filedata;
```

Voids

In a void declaration, the variable is not named. The declaration is just void and nothing else. Void declarations can only occur in two places: union definitions and program definitions as the argument or result of a remote procedure.

Appendix

A

RPC Manual Pages

This appendix lists the RPC library calls in UNIX-style manual page format.

RPC Library Functions

The RPC library calls are listed in alphabetical order in this chapter.

Following a brief introductory statement summarizing its use, each function is described using the documentation style of UNIX.

The following table lists the basic components of each function and its description:

Function	Description
Synopsis	A synopsis of the function is given in C language format. The function prototype statement is listed showing all function arguments.
Description	A description of the function is given, including any special rules for specifying arguments, alternative uses of the function, and any results returned.
Parameters	Each parameter for the call is described.
Files	Any required include files are listed in this section.
See Also	References to related functions are given.

auth_destroy()

Destroy authentication information.

Synopsis

```
void auth_destroy(auth)
AUTH *auth;
```

Description

auth_destroy() is a macro that destroys the authentication information associated with auth. Destruction usually involves deallocation of private data structures. The use of auth is undefined after calling auth_destroy(). This routine is called indirectly based on the pointer to the routine passed in the struct AUTH. This routine may also be called using the upper-case AUTH_DESTROY().

Parameters

auth	RPC authentication handle.

Files

rpc.h - RPC include file.

See Also

authnone_create(), authunix_create(), authunix_create_default().

authnone_create()

Create RPC authentication handle.

Synopsis

```
AUTH *authnone_create().
```

Description

authnone_create() creates and returns an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default used by RPC.

Files

rpc.h - RPC include file.

See Also

auth_destroy(), authunix_create(), authunix_create_default().

authunix_create()

Create RPC authentication handle.

Synopsis

```
AUTH    *authunix_create(host, uid, gid, len, aup_gids)
char    *host;
int     uid, gid, len, *aup_gids;
```

Description

authunix_create() creates and returns an RPC authentication handle that contains authentication information. It is easy to impersonate a user.

Parameters

host	The name of the machine on which the information was created.
uid	The user's user ID.
gid	The user's current group ID.
len	Refers to a counted array of groups to which the user belongs.

Files

rpc.h - RPC include file.

See Also

auth_destroy(), authnone_create(), authunix_create_default().

authunix_create_default()

Call authunix with default parameters.

Synopsis

```
AUTH *authunix_create_default()
```

Description

authunix_create_default() calls authunix_create() with the appropriate parameters.

Files

rpc.h - RPC include file.

See Also

auth_destroy(), authnone_create(), authunix_create()

callrpc()

Call remote procedure.

Synopsis

```
int callrpc(host, prognum, versnum, procnum, inproc, in,
             outproc, out)
char         *host;
u_long       prognum, versnum, procnum;
char         *in, *out;
xdrproc_t    inproc, outproc;
```

Description

callrpc() calls the remote procedure associated with program number, version number, and remote procedure on the machine host. This routine returns zero if it succeeds, or the value of enum clnt_stat cast to an integer if it fails. The routine clnt_perrno() is handy for translating failure statuses into messages.

Calling remote procedures with this routine uses UDP/IP as a transport; see clntudp_create() for restrictions. You do not have control of time-outs or authentication using this routine.

Parameters

host	Machine name.
prognum	Program number.
versnum	Version number.
procnum	Remote procedure.
inproc	XDR routine used to encode the procedure's parameters.
in	The address of the procedure's arguments.
outproc	XDR routine used to decode the procedure's results.
out	Address at which to place the result(s).

Files

rpc.h - RPC include file.

See Also

clnt_stat(), clnt_perrno(), clnttcp_create(), clntudp; create().

clnt_broadcast()

Broadcast RPC call message.

Note: This routine exists but it currently returns failure (-1).

Synopsis

```
enum clnt_stat clnt_broadcast(prognum, versnum, procnum,
                    inproc, in, outproc, out, eachresult)
u_long          prognum, versnum, procnum;
char            *in, *out;
xdrproc_t       inproc, outproc;
resultproc_t    eachresult;
```

Description

clnt_broadcast() is like callrpc(), except the call message is broadcast to all locally connected broadcast nets.

Each time it receives a response, this routine calls eachresult(), whose form is:

```
eachresult(out, addr)
char    *out;
struct   sockaddr_in *addr;
```

out	The same as out passed to clnt_broadcast(), except that the remote procedure's output is decoded there.
addr	Points to the address of the machine that sent the results.

If eachresult() returns zero, clnt_broadcast() waits for more replies; otherwise it returns with appropriate status.
Note that broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

Parameters

prognum	Program number.
versnum	Version number.
procnum	Remote procedure.
inproc	XDR routine used to encode the procedure's parameters.
in	The address of the procedure's arguments.
outproc	XDR routine used to decode the procedure's results.
out	The address of where to place the result(s).
eachresult	Routine called on response receipt (see above).

Files

rpc.h - RPC include file.

pmapclnt.h - Portmapper include file.

See Also

callrpc()

clnt_call()

Call remote procedure.

Synopsis

```
enum clnt_stat clnt_call(clnt, procnum, inproc, in, outproc,
                         out, timeout)
CLIENT            *clnt;
ulong             procnum;
xdrproc_t         inproc, outproc;
char              *in, *out;
struct timeval    timeout;
```

Description

clnt_call() is a macro that calls the remote procedure procnum associated with the client handle clnt which is obtained with an RPC client creation routine such as clnt_create(). This routine is called indirectly based on the pointer to the routine passed in the struct CLIENT. This call may also be called using the upper-case CLNT_CALL().

Parameters

clnt	Client handle.
procnum	Remote procedure.
inproc	XDR routine used to encode the procedure's parameters.
in	The address of the procedure's argument(s).
outproc	XDR routine used to decode the procedure's results.
out	Address of where to place the result(s).
timeout	Time allowed for results to come back.

Files

rpc.h - RPC include file.

See Also

clnt_stat(), clnt_perrno(), clnttcp_create(), clntudp_create()

clnt_control()

Change or receive information about client object.

Synopsis

```
bool_t clnt_control(cl, req, info)
CLIENT       *cl;
int          req;
char         *info
```

Description

clnt_control() is a macro used to change or retrieve various information about a client object. req indicates the type of operation, and info is a pointer to the information. For both UDP and TCP, the supported values of req and their argument types and what they do are:

```
CLSET_TIMEOUT  struct timeval   set total timeout
CLGET_TIMEOUT   struct timeval     get total timeout
```

Note: If you set the timeout using clnt_control() the timeout parameter passed to clnt_call() will be ignored in all future calls.

```
CLGET_SERVER_ADDR   struct sockaddr   get server's address
```

The following operations are valid for UDP only:

```
CLSET_RETRY_TIMEOUT   struct timeval   set the retry timeout
CLGET_RETRY_TIMEOUT   struct timeval   get the retry timeout
```

The retry timeout is the time that UDP RPC waits for the server to reply before retransmitting the request.

This routine is called indirectly based on the pointer to the routine passed in the struct CLIENT. This call may also be called using the upper-case CLNT_CONTROL().
This routine returns TRUE on success and FALSE on failure.

Parameters

cl	Client.
req	Indicates the type of operation.
info	A pointer to the information.

Files

rpc.h - RPC include file.

See Also

clnt_call()

clnt_create()

Client creation routine.

Synopsis

```
CLIENT     *clnt_create(host, prognum, versnum, proto)
char       *host;
u_long     prognum, versnum;
char      *proto;
```

Description

clnt_create() is a generic client creation routine. Default time-outs are set, but can be modified using clnt_control().

Using UDP has its shortcomings. Since UDP-based RPC messages can only hold up to eight KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

Parameters

host	Identifies the name of the remote host where the server is located.
prognum	Remote program number.
versnum	Remote version number.
proto	Indicates which kind of transport protocol to use. The currently supported values for this field are udp and tcp.

Files

rpc.h - RPC include file.

See Also

clnt_control(), clnt_destroy(), clnttcp_create(), clntudp_create()

clnt_destroy()

Destroy client's RPC handle.

Synopsis

```
void clnt_destroy(clnt)
CLIENT   *clnt;
```

Description

clnt_destroy() is a macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including clnt itself. Use of clnt is undefined after calling clnt_destroy(). If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open. This routine is called indirectly based on the pointer to the routine passed in the struct CLIENT. This call may also be called using the upper-case CLNT_DESTROY().

Parameters

clnt	Client handle.

Files

rpc.h - RPC include file.

See Also clnt_stat(), clnt_perrno(), clntudp_create()

clnt_freeres()

Free data allocated by result decoding.

Synopsis

```
bool_t clnt_freeres(clnt, outproc, out)
CLIENT      *clnt;
xdrproc_t   outproc;
char        *out;
```

Description

clnt_freeres() is a macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. This routine returns TRUE if the results were successfully freed, and FALSE otherwise. This routine is called indirectly based on the pointer to the routine passed in the struct CLIENT. This routine may also be called using the upper-case CLNT_FREERES().

Parameters

clnt	Client.
outproc	XDR routine describing the results.
out	Address of the results.

Files rpc. h - RPC include file.

See Also clnt_call()

clnt_geterr()

Get error structure.

Synopsis

```
void clnt_geterr(clnt, errp)
CLIENT          *clnt;
struct rpc_err     *errp;
```

Description

clnt_geterr() is a macro that copies the error structure out of the client handle to the structure at address errp. This routine is called indirectly based on the pointer to the routine passed in the struct CLIENT. This routine may also be called using the upper-case CLNT_GETERR().

Parameters

clnt	Client.
errp	Address of the error structure.

Files rpc.h - RPC include file.

See Also clnt_call(), clnt_call()

clnt_pcreateerror()

Print error about client creation.

Synopsis

```
void clnt_pcreateerror(str)
char       *str;
```

Description

clnt_pcreateerror() prints a message via the rpclog() facility indicating why a client RPC handle could not be created. The message is prepended with string str and a colon. Used when a clnt_create(), clntraw_create(), clnttcp_create(), or clntudp_create() call fails.

Parameters

str string to prepend

Files

rpc.h - RPC include file.

See Also

clnt_create(), clntraw_create(), clnttcp_create(), clntudp_create(), and clnt_spcreateerror()

clnt_perrno()

Print standard error.

Synopsis

```
void clnt_perrno(stat)
enum clnt_stat     stat;
```

Description

clnt_perrno() prints via the rpclog() facility, a standard error corresponding to the condition indicated by stat. Used after callrpc().

Parameters

stat Error indication.

Files

rpc.h - RPC include file.

See Also

callrpc(), clnt_call(), clnt_perror(), clnt_sperror(), and clnt_sperror()

clnt_perror()

Print message for why RPC call failed.

Synopsis

```
void clnt_perror(clnt, str)
CLIENT        *clnt;
char          *str;
```

Description

clnt_perror() prints a message via the rpclog() facility indicating why an RPC call failed. The message is prepended with string str and a colon. Used after clnt_call() or callrpc().

Parameters

clnt	Handle used to do the call.
str	String prepended to message.

Files

rpc.h - RPC include file.

See Also

clnt_call(), callrpc(), clnt_perrno(), clnt_sperrno(), and clnt_sperror()

clnt_specreaterror()

Returns a string for why RPC handle could not be created.

Synopsis

```
char *clnt_spcreateerror(str)
char *str;
```

Description

clnt_spcreateerror() returns a string indicating why a client RPC handle could not be created. clnt_spcreateerror() is like clnt_pcreateerror(), except that it returns a string instead of using the rpclog() facility.

Note: Returns pointer to static data that is overwritten on each call.

Parameters

str	String to prepend to message.

Files

rpc.h - RPC include file.

See Also

clnt_pcreateerror()

clnt_sperrno()

Returns a string for why an RPC call failed.

Synopsis

```
char *clnt_sperrno(stat)
enum clnt_stat      stat;
```

Description

clnt_sperrno() takes the same arguments as clnt_perrno(), but instead of sending a message to the rpclog() facility indicating why an RPC call failed, returns a pointer to a string which contains the message. The string ends with a newline.

clnt_sperrno() is used instead of clnt_perrno() if the program does not want to use the rpclog() facility, or if a message format different than that supported by clnt_perrno() is to be used.

Note: Unlike clnt_sperror() and clnt_spcreateerror(), clnt_sperrno() does not return a pointer to static data, so the result will not get overwritten on each call.

Parameters

stat	Error condition.

Files

rpc.h - RPC include file.

See Also

clnt_perrno(), clnt_sperror(), and clnt_spcreateerror()

clnt_sperror()

Returns a string for why an RPC call failed.

Synopsis

```
char *clnt_sperrno(rpch, str)
CLIENT    *rpch;
char      *str;
```

Description

clnt_sperror( )is like clnt_perror(), except that, like clnt_sperrno(), it returns a string instead of using the rpclog() facility.

Note: Returns a pointer to static data that is overwritten on each call.

Parameters

rpch	Handle.
str	String.

Files

rpc.h - RPC include file.

See Also

clnt_perrno(), and clnt_perror()

clntraw_create()

Creates an RPC client.

Synopsis

```
CLIENT   *clntraw_create(prognum, versnum)
u_long   prognum, versnum;
```

Description

clntraw_create() creates an RPC client for the remote program prognum, version versnum. The transport used to pass messages to the service is actually a buffer within the task's address space, so the corresponding RPC server should live in the same address space; see svcraw_create(). This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any network interference. This routine returns NULL if it fails.

Parameters

prognum	Remote program.
versnum	Version.

Files

rpc.h - RPC include file.

See Also

svcraw_create()

clnttcp_create()

Creates an RPC client that uses TCP.

Synopsis

```
CLIENT *clnttcp_create(addr, prognum, versnum, sockp,
                       sendsz, recvsz)
struct sockaddr_in     *addr;
u_long                  prognum, versnum;
int                    *sockp;
u_int                   sendsz, recvsz;
```

Description

clnttcp_create() creates an RPC client for the remote program prognum, version versnum; the client uses TCP/IP as a transport. The remote program is located at Internet address *addr. If addr->sin_port is zero, then it is set to the actual port that the remote program is listening on (the remote portmap service is consulted for this information). The parameter sockp is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets sockp. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters sendsz and recvsz; values of zero choose suitable defaults. This routine returns NULL if it fails.

Parameters

addr	Internet address.
prognum	Remote program.
versnum	Version.
sockp	Pointer to a socket.
sendsz	Size of send buffer.
recvsz	Size of receive buffer.

Files

rpc.h - RPC include file.

See Also

clntudp_create()

clntudp_create()

Creates an RPC client which uses TCP.

Synopsis

```
CLIENT *clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in      *addr;
u_long                   prognum, versnum;
struct timeval           wait;
int                     *sockp;
```

Description

clntudp_create() creates an RPC client for the remote program prognum, version versnum; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr. If addr->sin_port is zero, then it is set to the actual port that the remote program is listening on (the remote portmap service is consulted for this information). The parameter sockp is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets sockp. The UDP transport resends the call message in intervals of wait time until a response is received or until the call times out. The total time for the call to time out is specified by clnt_call().

Since UDP-based RPC messages can only hold up to 8K of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

Parameters

addr	Internet address.
prognum	Remote program.
versnum	Version.
wait	Time interval to resend message.
sockp	Socket pointer.

Files

rpc.h - RPC include file.

See Also

clnttcp_create()

get_myaddress()

Get machine's IP address.

Synopsis

```
void get_myaddress(addr)
struct sockaddr_in  *addr;
```

Description

get_myaddress() returns the machine's IP address in *addr, without consulting the library routines that deal with /etc/hosts. The port number is always set to htons(PMAPPORT).

Parameters

addr Internet address.

Files

rpc. h - RPC include file.

See Also

htons()

getrpcbyname()

Get RPC entry by name.

Synopsis

```
struct rpcent *getrpcbyname(name)
char *name;
```

Description

getrpcbyname() returns a pointer to an object with the following structure containing the information returned by the Domain Name Resolver (DNR).

```
structrpcent{
    char  *r_name;          /* name of server for this rpc program */
    char  **r_aliases;      /* alias list */
    long  r_number;         /* rpc program number */
};
```

r_name Name of the server for this RPC program.

r_aliases Zero terminated list of alternate names for the RPC program.

r_number RPC program number for this service.

getrpcbyname() queries DNR with a DFRPCBYN request.

Parameters

name Name of server.

Files

rpc.h - RPC include file.

See Also

getrpcbynumber()

getrpcbynumber()

Get RPC entry by number.

Synopsis

```
struct rpcent *getrpcbynumber(number)
int number;
```

Description

getrpcbynumber() queries the Domain Name Resolver (DNR) and returns a pointer to an object with the following structure:

```
structrpcent{
    char  *r_name;         /* name of server for this rpc program */
    char  **r_aliases;     /* alias list */
    long  r_number;        /* rpc program number */
    };
```

r_name — Name of the server for this RPC program.

r_aliases — A zero terminated list of alternate names for the RPC program.

r_number — RPC program number for this service.

getrpcbynumber() queries DNR with a DFRPCBYV request.

Parameters

number — RPC program number.

Files

rpc.h - RPC include file.

See Also

getrpcbyname()

mvs_svc_run()

Call the appropriate service routine for RPC requests.

Synopsis

```
int mvs_svc_run(ecblistp, ecbcount)
unsigned long    **ecblistp;
int                ecbcount;
```

Description

mvs_svc_run() waits for RPC requests to arrive, and calls the appropriate service procedure using svc_getreq() when one arrives. mvs_svc_run() is similar to svc_run() but can wait on an ECB list in addition to the socket wait. This call returns if any ECB is posted. This procedure is usually waiting for a select() system call to return. mvs_svc_run() also returns if the API shuts down or encounters a system error.

Parameters

ecblistp	Pointer to ECB list.
ecbcount	ECB count to wait on.

Files

rpc.h - RPC include file.

See Also

svc_run()

pmap_getmaps()

Get list of port mappings.

Synopsis

```
struct pmaplist *pmap_getmaps(addr)
struct sockaddr_in    *addr;
```

Description

pmap_getmaps() is a user interface to the portmap service, which returns a list of the current RPC program-to-port mappings on the host located at IP address *addr. This routine can return NULL. The rpcinfo utility uses this routine.

Parameters

addr	Internet address.

Files

rpc.h - RPC include file.

pmapclnt.h - Portmapper include file.

See Also

pmap_getport(), pmap_set(), pmap_unset()

pmap_getport()

Get port number for a service.

Synopsis

```
u_short pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in   *addr;
u_long               prognum, versnum, protocol;
```

Description

pmap_getmaps() is a user interface to the portmap service, which returns the port number on which a service waits that supports program number prognum, version versnum, and speaks the transport protocol associated with protocol. The value of protocol is most likely IPPROTO_UDP or IPPROTO_TCP. A return value of zero means that the mapping does not exist or that the RPC system failed to contact the remote portmap service. In the latter case, the global variable rpc_createerr() contains the RPC status.

Parameters

addr	Internet address.
prognum	Remote program number.
versnum	Version number.
protocol	Transport protocol.

Files

rpc.h - RPC include file.

pmapclnt.h - Portmapper include file.

See Also

pmap_getmaps(), pmap_set(), pmap_unset()

pmap_rmtcall()

Tell portmapper to make an RPC call.

Synopsis

```
enum clnt_stat pmap_rmtcall(addr, prognum, versnum, procnum,
            inproc, in, outproc, out, timeout, portp)
struct sockaddr_in     *addr;
u_long                  prognum, versnum, procnum;
char                   *in, *out;
xdrproc_t               inproc, outproc;
struct timeval          timeout;
u_long                 *portp;
```

Description

pmap_rmtcall() is a user interface to the portmap service, which instructs portmap on the host at IP address *addr to make an RPC call on your behalf to a procedure on that host. The parameter *portp will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in callrpc() and clnt_call(). This procedure should be used for a ping and nothing else. See also clnt_broadcast().

Parameters

addr	Internet address.
prognum	Remote program number.
versnum	Version number.
procnum	Procedure number.
inproc	XDR procedure used to encode the procedure's parameters.
in	The address of the procedure's arguments.
outproc	XDR procedure used to decode the procedure's results.
out	Address of where to place the result(s).
timeout	Time allowed for results to come back.
portp	Pointer to program's port number.

Files

rpc.h - RPC include file.

pmapclnt.h - Portmapper include file.

See Also

pmap_getmaps(), pmap_getport(), pmap_set(), pmap_unset()

pmap_set()

Set portmapping.

Synopsis

```
bool_t pmap_set(prognum, versnum, protocol, port)
u_long    prognum, versnum, protocol;
u_long    port;
```

Description

pmap_set() is a user interface to the portmap service, which establishes a mapping between the triple (prognum, versnum, protocol) and port on the machine's portmap service. The value of protocol is most likely IPPROTO_UDP or IPPROTO_TCP. This routine returns TRUE if it succeeds, FALSE otherwise. It is automatically done by svc_register().

Parameters

prognum	Program number.
versnum	Version number.
protocol	Transport protocol.
port	Program's port number.

Files

rpc.h - RPC include file.

pmapclnt.h - Portmapper include file.

See Also

pmap_getmaps(), pmap_getport(), pmap_unset()

pmap_unset()

Unset portmapping.

Synopsis

```
bool_t pmap_unset(prognum, versnum)
u_long  prognum, versnum;
```

Description

pmap_unset() is a user interface to the portmap service, which destroys all mapping between the triple (prognum, versnum,*) and ports on the machine's portmap service. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

prognum	Program number.
versnum	Version number.

Files

rpc.h - RPC include file.

pmapclnt.h - Portmapper include file.

See Also

pmap_getmaps(), pmap_getport(), pmap_set()

registerrpc()

Register a procedure with RPC.

Synopsis

```
int registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
u_long            prognum, versnum, procnum;
char             *(*procname)();
xdrproc_t          inproc, outproc;
```

Description

registerrpc() registers procedure procname with the RPC service package. If a request arrives for program prognum, version versnum, and procedure procnum, procname is called with a pointer to its parameter(s); procname should return a pointer to its static result(s); inproc is used to decode the parameters; outproc is used to encode the results. This routine returns zero if the registration succeeded, -1 otherwise.

Remote procedures registered in this form are accessed using the UDP/IP transport; see svcudp_create() for restrictions.

Parameters

prognum	Program number.
versum	Version number.
procnum	Procedure number.
procname	Procedure name
inproc	XDR procedure used to decode the procedure's parameters.
outproc	XDR procedure used to encode the procedure's results.

Files

rpc.h - RPC include file.

See Also

svcudp_create()

rpc_createerr

Global variable for unsuccessful client creation.

Synopsis

```
struct rpc_createerr rpc_createerr
```

Description

rpc_createerr is a global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine clnt_pcreateerror() to print the reason why.

Files

rpc.h - RPC include file.

See Also

clnt_pcreateerror()

svc_destroy()

Destroy RPC transport handle.

Synopsis

```
void svc_destroy(xprt)
SVCXPRT    *xprt;
```

Description

svc_destroy() is a macro that destroys the RPC service transport handle, xprt. Destruction usually involves deallocation of private data structures, including xprt itself. Use of xprt is undefined after calling this routine. This routine is called indirectly based on the pointer to the routine passed in the struct SVCXPRT. This routine may also be called using the upper-case SVC_DESTROY().

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getcaller(), svc_getreqset(), svc_getreq(), svc_register(),svc_run(), svc_sendreply() svc_unregister(), svcudp_create(), svctcp_create()

svc_fdset

Global variable for RPC's file descriptor bit mask.

Synopsis

```
fd_set svc_fdset;
```

Description

svc_fdset is a global variable reflecting the RPC service side's read file descriptor bit mask. It is suitable as a parameter to the select system call. This is only of interest if a service implementor does not call svc_run(), but rather does his own asynchronous event processing. This variable may change after calls to svc_getreqset() or any creation routines.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getcaller(), svc_getreqset(), svc_getreq(), svc_register(),svc_run(), svc_sendreply() svc_unregister(), svctcp_create(), svcudp_create()

svc_freeargs()

Free data allocated by svc_getargs argument decoding.

Synopsis

```
bool_t svc_freeargs(xprt, inproc, in)
SVCXPRT      *xprt;
xdrproc_t     inproc;
char         *in;
```

Description

svc_freeargs() is a macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using svc_getargs(). This routine returns TRUE if the results were successfully freed, and FALSE otherwise. This routine is called indirectly based on the pointer to the routine passed in the struct SVCXPRT. This routine may also be called with the upper-case SVC_FREEARGS().

Parameters

xprt	RPC service transport handle.
inproc	Used to encode the procedure's parameters.
in	Address of the procedure's arguments.

Files

rpc.h - RPC include file.

See Also

svc_getargs(), svc_getcaller(), svc_getreqset(), svc_getreq(), svc_register(),svc_run(), svc_sendreply() svc_unregister()

svc_getargs()

Decode RPC request arguments.

Synopsis

```
bool_t svc_getargs(xprt, inproc, in)
SVCXPRT     *xprt;
xdrproc_t    inproc;
char        *in;
```

Description

svc_getargs() is a macro that decodes the arguments of an RPC request associated with the RPC service transport handle, xprt. The parameter in is the address where the arguments will be placed; inproc is the XDR routine used to decode the arguments. This routine returns TRUE if decoding succeeds, and FALSE otherwise. This routine is called indirectly based on the pointer to the routine passed in the struct SVCXPRT. This routine may also be called with the upper-case SVC_GETARGS().

Parameters

xprt	RPC service transport handle.
inproc	Used to encode the procedure's parameters.
in	Address of the procedure's arguments.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getcaller(), svc_getreqset(), svc_getreq(), svc_register(),svc_run(), svc_sendreply() svc_unregister()

svc_getcaller()

Get network address of caller.

Synopsis

```
struct sockaddr_in *svc_getcaller(xprt)
SVCXPRT    *xprt;
```

Description

svc_getcaller() is the approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, xprt.

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getreqset(), svc_getreq(), svc_register(),svc_run(), svc_sendreply() svc_unregister()

svc_getreq()

Service an RPC request on a socket.

Synopsis

```
svc_getreq(rdfds)
int rdfds;
```

Description

svc_getreq() is similar to svc_getreqset(). This interface is obsoleted by svc_getreqset().

Parameters

rdfds Read file descriptor bit mask.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getcaller(), svc_getreqset(), svc_register(),svc_run(), svc_sendreply() svc_unregister()

svc_getreqset()

Service an RPC request that arrived on a socket.

Synopsis

```
svc_getreqset(rdfds)
fd_set   *rdfds;
```

Description

svc_getreqset() is only of interest if a service implementor does not call svc_run(), but instead implements custom asynchronous event processing. It is called when the select() system call has determined that an RPC request has arrived on some RPC socket(s); rdfds is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of rdfds have been serviced.

Parameters

rdfds Read file descriptor bit mask.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getcaller(), svc_getreq(), svc_register(), svc_run(), svc_sendreply(), svc_unregister()

svc_register()

Register procedure with service dispatch procedure.

Synopsis

```
bool_t svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT    *xprt;
u_long      prognum, versnum;
void        (*dispatch) ();
u_long      protocol;
```

Description

svc_register() associates prognum and versnum with the service dispatch procedure, dispatch. If protocol is zero, the service is not registered with the portmap service. If protocol is non-zero, then a mapping of the triple [prognum, versnum, protocol] to xprt->xp_port is established with the local portmap service (generally protocol is zero, IPPROTO_UDP or IPPROTO_TCP). The procedure dispatch has the following form:

```
dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

The svc_register() routine returns TRUE if it succeeds, and FALSE otherwise.

Parameters

xprt	RPC service transport handle.
prognum	Program number.
versnum	Version number.
dispatch	Service dispatch procedure.
protocol	Transport protocol.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getcaller(), svc_getreqset(), svc_getreq(), svc_run(), svc_sendreply() svc_unregister()

svc_run()

Call the appropriate service routine for RPC requests.

Synopsis

```
void svc_run()
```

Description

svc_run() waits for RPC requests to arrive, and calls the appropriate service procedure using svc_getreq() when one arrives. This procedure is usually waiting for a select() system call to return. svc_run() does not return unless the API shuts down or encounters a system error.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getcaller(), svc_getreqset(), svc_getreq(), svc_register(), svc_sendreply() svc_unregister()

svc_sendreply()

Send results of remote procedure call.

Synopsis

```
bool_t svc_sendreply(xprt, outproc, out)
SVCXPRT       *xprt;
xdrproc_ t     outproc;
char          *out;
```

Description

svc_sendreply() is called by an RPC service's routine to send the results of a remote procedure call. The parameter xprt is the request's associated transport handle; outproc is the XDR routine which is used to encode the results; and out is the address of the results. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xprt	RPC service transport handle.
outproc	XDR routine used to decode the procedure's results.
out	Address of where to place the result(s).

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getcaller(), svc_getreqset(), svc_getreq(), svc_register(), svc_sendreply() svc_unregister()

svc_unregister()

Remove mapping to dispatch routines.

Synopsis

```
void svc_unregister(prognum, versnum)
u_long    prognum, versnum;
```

Description

svc_unregister() removes all mapping of the double [prognum, versnum] to dispatch routine, and of the triple [prognum, versnum,*] to port number.

Parameters

prognum	Program number.
versnum	Version number.

Files

rpc.h - RPC include file.

See Also

svc_freeargs(), svc_getargs(), svc_getcaller(), svc_getreqset(), svc_getreq(), svc_register(), svc_sendreply()

svcerr_weakauth()

Called when insufficient authentication parameters are given.

Synopsis

void svcerr_weakauth(xprt)
`SVCXPRT *xprt;`

Description

svcerr_weakauth() is called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls svcerr_auth(xprt, AUTH_TOOWEAK).

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

svcerr_auth(), svcerr_decode(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr()

svcerr_auth()

Called after an authentication error.

Synopsis

```
void svcerr_auth(xprt, why)
SVCXPRT          *xprt
enum auth_stat    why;
```

Description

svcerr_auth() is called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

Parameters

xprt	RPC service transport handle.
why	Error.

Files

rpc.h - RPC include file.

See Also

svcerr_decode(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth()

svcerr_decode()

Called for parameter decoding error.

Synopsis

```
void svcerr_decode(xprt)
SVCXPRT *xprt;
```

Description

svcerr_decode() is called by a service dispatch routine that cannot successfully decode its parameters. See also svc_getargs().

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

svcerr_auth(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth()

svcerr_noproc()

Called for procedure number error.

Synopsis

```
void svcerr_noproc(xprt)
SVCXPRT *xprt;
```

Description

svcerr_noproc() is called by a service dispatch routine that does not implement the procedure number that the caller requests.

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

svcerr_auth(), svcerr_decode(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth()

svcerr_noprog()

Called when program is not registered.

Synopsis

```
void svcerr_noprog(xprt)
SVCXPRT *xprt;
```

Description

svcerr_noprog() is called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

svcerr_auth(), svcerr_decode(), svcerr_noproc(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth()

svcerr_progvers()

Called when program version is not registered.

Synopsis

void svcerr_progvers(xprt)
SVCXPRT *xprt;

Description

svcerr_progvers() is called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

svcerr_auth(), svcerr_decode(), svcerr_noproc(), svcerr_noprog(),svcerr_systemerr(), svcerr_weakauth()

svcerr_systemerr()

Called when a system error is detected.

Synopsis

```
void svcerr_systemerr(xprt)
SVCXPRT *xprt;
```

Description

svcerr_systemerr() is called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

svcerr_auth(), svcerr_decode(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_weakauth()

svcfd_create()

Create a service on top of any open descriptor.

Synopsis

```
void svcfd_create(fd, sendsize, recvsize)
int     fd;
u_int   sendsize;
u_int   recvsize;
```

Description

svcfd_create() creates a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. sendsize and recvsize indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

Parameters

fd	Descriptor.
endsize	Size of send buffer.
recvsize	Size of receive buffer.

Files

rpc.h - RPC include file.

See Also

svctcp_create(), svcudp_create()

svcraw_create()

Create an RPC service transport.

Synopsis

```
SVCXPRT *svcraw_create()
```

Description

svcraw_create() creates an RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see clntraw_create(). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

Files

rpc.h - RPC include file.

See Also

clntraw_create()

svctcp_create()

Create a TCP/IP based service transport.

Synopsis

```
SVCXPRT *svctcp_create(sock, send_buf_size, recv_buf_size)
int    sock;
u_int  send_buf_size, recv_buf_size;
```

Description

svctcp_create() creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket sock, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, xprt->xp_sock is the transport's socket descriptor; xprt->xp_port is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of zero choose suitable defaults.

Parameters

sock	Socket.
send_buf_size	Size of send buffer.
recv_buf_size	Size of receive buffer.

Files

rpc.h - RPC include file.

See Also

svcudp_create()

svcudp_create()

Create a UDP/IP based service transport.

Synopsis

```
SVCXPRT *svcudp_create(sock)
int sock;
```

Description

svcudp_create() creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket sock, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, xprt->xp_sock is the transport's socket descriptor; whereas the field xprt->xp_port is the transport's port number. This routine returns NULL if it fails.

Since UDP-based RPC messages can only hold up to 8K of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

Parameters

sock	Socket.

Files

rpc.h - RPC include file.

See Also

svctcp_create()

xdr_accepted_reply()

Encode RPC reply messages.

Synopsis

```
bool_t xdr_accepted_reply(xdrs, ar)
XDR                     *xdrs;
struct accepted_reply   *ar;
```

Description

xdr_accepted_reply() is used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package. This routine returns TRUE if successful, otherwise it returns FALSE.

Parameters

xdrs	XDR structure.
ar	Reply.

Files

rpc.h - RPC include file.

See Also

xdr_authunix_parms(), xdr_callhdr(), xdr_callmsg(), xdr_opaque_auth(), xdr_pmap(), xdr_pmaplist(), xdr_rejected_reply(), xdr_replymsg()

xdr_authunix_parms()

Describe UNIX credentials.

Synopsis

```
bool_t xdr_authunix_parms(xdrs, aupp)
XDR                      *xdrs;
struct authunix_parms    *aupp;
```

Description

xdr_authunix_parms() is used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package. This routine returns TRUE if successful, otherwise it returns FALSE.

Parameters

xdrs	XDR structure.
aupp	UNIX credentials.

Files

rpc.h - RPC include file.

See Also

xdr_accepted_reply(), xdr_callhdr(), xdr_callmsg(), xdr_opaque_auth(), xdr_pmap(), xdr_pmaplist(), xdr_rejected_reply(), xdr_replymsg()

xdr_callhdr()

Describe RPC call header messages.

Synopsis

```
void xdr_callhdr(xdrs, chdr)
XDR             *xdrs;
struct rpc_msg   *chdr;
```

Description

xdr_callhdr() is used for describing RPC call header messages. It encodes the static part of the call message header in the XDR language format. It includes information such as transaction ID, RPC version number, program number, and version number. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

Parameters

xdrs	XDR structure.
chdr	Call header message.

Files

rpc.h - RPC include file.

See Also

xdr_accepted_reply(), xdr_authunix_parms(), xdr_callmsg(), xdr_opaque_auth(), xdr_pmap(), xdr_pmaplist(), xdr_rejected_reply(), xdr_replymsg()

xdr_callmsg()

Describe RPC call messages.

Synopsis

```
bool_t xdr_callmsg(xdrs, cmsg)
XDR               *xdrs;
struct rpc_msg    *cmsg;
```

Description

xdr_callmsg() is used for describing RPC call messages. It includes all the RPC call information such as transaction ID, RPC version number, program number and version number, authentication information, etc. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

This routine returns TRUE if successful, FALSE otherwise

Parameters

xdrs	XDR structure.
cmsg	Call message.

Files

rpc.h - RPC include file.

See Also

xdr_accepted_reply(), xdr_authunix_parms(), xdr_callhdr(), xdr_opaque_auth(), xdr_pmap(), xdr_pmaplist(), xdr_rejected_reply(), xdr_replymsg()

xdr_opaque_auth()

Describe RPC authentication information message.

Synopsis

```
bool_t xdr_opaque_auth(xdrs, ap)
XDR                   *xdrs;
struct opaque_auth    *ap;
```

Description

xdr_opaque_auth() is used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package. This routine returns TRUE if successful, FALSE otherwise.

Parameters

xdrs	XDR structure.
ap	Authentication information message.

Files

rpc.h - RPC include file.

See Also

xdr_accepted_reply(), xdr_authunix_parms(), xdr_callhdr(), xdr_callmsg(), xdr_pmap(), xdr_pmaplist(), xdr_rejected_reply(), xdr_replymsg()

xdr_pmap()

Describe parameters to portmap procedures.

Synopsis

```
bool_t xdr_pmap(xdrs,regs)
XDR             *xdrs;
struct pmap     *regs;
```

Description

xdr_pmap() is used for describing parameters to various portmap procedures, externally. This routine is useful for users who wish to generate RPC-style messages without using the pmap package.

This routine returns TRUE if successful, FALSE otherwise.

Parameters

xdrs	XDR structure.
regs	Portmap parameters.

Files

rpc.h - RPC include file.

pmapport.h - Portmap include file.

See Also

xdr_accepted_reply(), xdr_authunix_parms(), xdr_callhdr(), xdr_callmsg(), xdr_opaque_auth(), xdr_pmaplist(), xdr_rejected_reply(), xdr_replymsg()

xdr_pmaplist()

Describe list of portmappings.

Synopsis

```
bool_t xdr_pmaplist(xdrs, rp)
XDR                *xdrs;
struct pmaplist    **rp;
```

Description

xdr_pmaplist() is used for describing a list of port mappings, externally. This routine is useful for users who wish to generate RPC-style messages without using the pmap interface.

This routine returns TRUE if successful, FALSE otherwise.

Parameters

xdrs	XDR structure.
rp	Port mapping list.

Files

rpc.h -RPC include file.

pmapport.h - portmap include file.

See Also

xdr_accepted_reply(), xdr_authunix_parms(), xdr_callhdr(), xdr_callmsg(), xdr_opaque_auth(), xdr_pmap(), xdr_rejected_reply(), xdr_replymsg()

xdr_rejected_reply()

Describe RPC reply messages.

Synopsis

```
bool_t xdr_rejected_reply(xdrs, rr)
XDR                       *xdrs;
struct rejected_reply     *rr;
```

Description

xdr_rejected_reply() is used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

This routine returns TRUE if successful, FALSE otherwise.

Parameters

xdrs	XDR structure.
rr	Reply message.

Files

rpc.h - RPC include file.

See Also

xdr_accepted_reply(), xdr_authunix_parms(), xdr_callhdr(), xdr_callmsg(), xdr_opaque_auth(), xdr_pmap(), xdr_pmaplist(), xdr_replymsg()

xdr_replymsg()

Describe RPC reply messages.

Synopsis

```
bool_t xdr_replymsg(xdrs, rmsg)
XDR               *xdrs;
struct rpc_msg    *rmsg;
```

Description

xdr_replymsg() is used for describing RPC reply messages. This reply could be an acceptance, rejection, or NULL. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

This routine returns TRUE if successful, FALSE otherwise.

Parameters

xdrs	XDR structure.
rms	Reply message.

Files

rpc.h - RPC include file.

See Also

xdr_accepted_reply(), xdr_authunix_parms(), xdr_callhdr(), xdr_callmsg(), xdr_opaque_auth(), xdr_pmap(), xdr_pmaplist(), xdr_rejected_reply()

xprt_register()

Register RPC service transport handle.

Synopsis

```
void xprt_register(xprt)
SVCXPRT *xprt;
```

Description

xprt_register() is used after RPC service transport handles are created, to register them with the RPC service package. This routine modifies the global variable svc_fds. Service implementors usually do not need this routine.

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

xprt_unregister()

xprt_unregister()

Unregister RPC service transport handle.

Synopsis

```
void xprt_unregister(xprt)
SVCXPRT *xprt;
```

Description

xprt_unregister() is used before an RPC service transport handle is destroyed, to unregister it with the RPC service package. This routine modifies the global variable svc_fds. Service implementors usually do not need this routine.

Parameters

xprt	RPC service transport handle.

Files

rpc.h - RPC include file.

See Also

xprt_unregister()

Appendix

B XDR Manual Pages

This appendix lists the XDR library calls in UNIX-style manual page format.

XDR Library Calls

The XDR library calls are listed in alphabetical order.

Following a brief introductory statement summarizing its use, each function is described using the documentation style of UNIX. The following table lists the basic components of each function description:

Call	Description
Synopsis	A synopsis of the function is given in C language format. The function prototype statement is listed showing all function arguments.
Description	A description of the function is given, including any special rules for specifying arguments, alternative uses of the function, and any results returned.
Parameters	Each parameter for the call is described.
Files	Any required include files are listed in this section.
See Also	References to related functions are given.

xdr_array()

Translate between arrays and their external representations.

Synopsis

```
bool_t xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR          *xdrs;
char        **arrp;
u_int        *sizep, maxsize, elsize;
xdrproc_t     elproc;
```

Description

xdr_array() is a filter primitive that translates between variable-length arrays and their corresponding external representations. The parameter arrp is the address of the pointer to the array, while sizep is the address of the element count of the array; this element count cannot exceed maxsize. The parameter elsize is the sizeof each of the array's elements, and elproc is an XDR filter that translates between the array elements' C form and their external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
arrp	Address of the pointer to the array.
sizep	Address of the element count of the array; this element count cannot exceed maxsize.
maxsize	Maximum element count.
elsize	Size of each of the array's elements.
elproc	XDR filter that translates between the array elements' C form and their external representation.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char().

xdr_bool()

Translate between booleans and their external representations.

Synopsis

```
bool_t xdr_bool(xdrs, bp)
XDR      *xdrs;
bool_t   *bp;
```

Description

xdr_bool() is a filter primitive that translates between booleans and their external representations. When encoding data, this filter produces values of either TRUE or FALSE. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
bp	Address of the Boolean.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char().

xdr_bytes()

Translate between counted byte strings and their external representations.

Synopsis

```
bool_t xdr_bytes(xdrs, sp, sizep, maxsize)
XDR       *xdrs;
char      **sp;
u_int      *sizep, maxsize;
```

Description

xdr_bytes() is a filter primitive that translates between counted byte strings and their external representations. The parameter sp is the address of the string pointer. The length of the string is located at sizep; strings cannot be longer than maxsize. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
sp	Address of the string pointer.
sizep	Length of the string.
maxsize	Maximum length of string.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char().

xdr_char()

Translate between C characters and their external representations.

Synopsis

```
bool_t xdr_char(xdrs, cp)
XDR     *xdrs;
char    *cp;
```

Description

xdr_char() is a filter primitive that translates between C characters and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Note: Encoded characters are not packed and occupy four bytes each. For arrays of characters, it is worthwhile to consider xdr_bytes(), xdr_opaque() or xdr_string().

Parameters

xdrs	XDR stream.
cp	Address of the character.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char(), xdr_opaque(), xdr_string().

xdr_destroy()

Destroy routine associated with XDR stream.

Synopsis

```
void xdr_destroy(xdrs)
XDR  *xdrs;
```

Description

xdr_destroy() is a macro that invokes the destroy routine associated with the XDR stream, xdrs. Destruction usually involves freeing private data structures associated with the stream. Using xdrs after invoking xdr_destroy() is undefined. This routine is called indirectly via the pointer stored in the structure XDR. This routine may also be called using the upper-case XDR_DESTROY().

Parameters

xdrs	XDR stream.

Files

rpc.h - RPC include file.

xdr_double()

Translate between C double precision numbers and their external representations.

Synopsis

```
bool_t xdr_double(xdrs, dp)
XDR      *xdrs;
double   *dp;
```

Description

xdr_double() is a filter primitive that translates between C double precision numbers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
dp	Address of double precision number.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char(), xdr_opaque(), xdr_string().

xdr_enum()

Translate between C enums and their external representations.

Synopsis

```
bool_t xdr_double(xdrs, ep)
XDR       *xdrs;
enum_t    *ep;
```

Description

xdr_enum() is a filter primitive that translates between C enums and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Note: Enums with the IBM C/370 compiler are scaled by the size of the maximum value defined for the enum. Therefore an enum may be a byte, two bytes or four bytes long. This routine assumes that an enum will be four bytes long.

Parameters

xdrs	XDR stream.
ep	Enum.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char(), xdr_opaque(), xdr_string().

xdr_float()

Translate between C floats and their external representations.

Synopsis

```
bool_t xdr_float(xdrs, fp)
XDR      *xdrs;
float    *fp;
```

Description

xdr_float() is a filter primitive that translates between C floats and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
fp	Enum.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char(), xdr_opaque(), xdr_string().

xdr_free()

Generic freeing routine.

Synopsis

```
void xdr_free(proc, objp)
xdrproc_t     proc;
char         *objp;
```

Description

xdr_free() is a generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. The pointer passed to this routine is not freed, but what it points to is freed (recursively), such that objects pointed to are also freed, for example, limited lists.

Parameters

xdrproc	XDR routine.
objp	Object to be freed.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char(), xdr_opaque(), xdr_string().

xdr_getpos()

Invoke get-position routine.

Synopsis

```
u_int xdr_getpos(xdrs)
XDR  *xdrs;
```

Description

xdr_getpos() is a macro that invokes the get-position routine associated with the XDR stream, xdrs. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this. This routine is called indirectly via the pointer stored in the structure XDR. This routine may also be called using the upper-case XDR_GETPOS().

Parameters

xdrs	XDR stream.

Files

rpc.h - RPC include file.

See Also

xdr_inline().

xdr_inline()

Invoke in-line routine.

Synopsis

```
xdr_inline(xdrs, len)
XDR    *xdrs;
int     len;
```

Description

xdr_inline() is a macro that invokes the in-line routine associated with the XDR stream, xdrs. The routine returns a pointer to a contiguous piece of the stream's buffer; len is the byte length of the desired buffer.

The pointer is cast to long *. This routine is called indirectly via the pointer stored in the structure XDR. This routine may also be called using the upper-case XDR_GETPOS().

xdr_inline() may return NULL (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

Parameters

xdrs	XDR stream.
len	Length of desired buffer.

Files

rpc.h - RPC include file.

See Also

xdr_getpos().

xdr_int()

Translate between C integers and their external representations.

Synopsis

```
bool_t xdr_int(xdrs, ip)
XDR    *xdrs;
int    *ip;
```

Description

xdr_int() is a filter primitive that translates between C integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
ip	Integer.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char()cc, xdr_opaque(), xdr_string().

xdr_long()

Translate between C long integers and their external representations.

Synopsis

```
bool_t xdr_long(xdrs, lp)
XDR     *xdrs;
long    *lp;
```

Description

xdr_long() is a filter primitive that translates between C long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
lp	Long.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char(), xdr_opaque(), xdr_string().

xdr_opaque()

Translate between fixed size opaque data and its external representation.

Synopsis

```
bool_t xdr_opaque(xdrs, cp, cnt)
XDR      *xdrs;
char     *cp;
u_int     cnt;
```

Description

xdr_opaque() is a filter primitive that translates between fixed size opaque data and its external representation. The parameter cp is the address of the opaque object, and cnt is its size in bytes. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
cp	Opaque object.
cn	Size of object.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char(), xdr_opaque(), xdr_string().

xdr_pointer()

Provide pointer chasing within structures.

Synopsis

```
bool_t xdr_pointer(xdrs, objpp, objsize, xdrobj)
XDR           *xdrs;
char         **objpp;
u_int          objsize;
xdrproc_t      xdrobj;
```

Description

xdr_pointer() provides pointer chasing within structures. xdr_pointer() is like xdr_reference() except that it serializes NULL pointers, whereas xdr_reference() does not. Thus, xdr_pointer() can represent recursive data structures, such as binary trees or linked lists. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
objpp	Address of object pointer.
objsize	Size of object.
xdrobj	XDR procedure that filters the structure between its C form and its external representation.

Files

rpc.h - RPC include file.

See Also

xdr_reference()

xdr_reference()

Provides pointer chasing within structures.

Synopsis

```
bool_t xdr_reference(xdrs, pp, size, proc)
XDR           *xdrs;
char         **pp;
u_int          size;
xdrproc_t      proc;
```

Description

xdr_reference() is a primitive that provides pointer chasing within structures. The parameter pp is the address of the pointer; size is the size of the structure that *pp points to; and proc is an XDR procedure that filters the structure between its C form and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

This routine does not understand NULL pointers. Use xdr_pointer() instead.

Parameters

xdrs	XDR stream.
pp	Address of the pointer.
size	Size of the structure that *pp points to.
proc	Size of the structure between its C and form and its external representation.

Files

rpc.h - RPC include file.

See Also

xdr_pointer().

xdr_setpos()

Invoke set position routine.

Synopsis

```
bool_t xdr_setpos(xdrs, pos)
XDR      *xdrs;
u_int     pos;
```

Description

xdr_setpos() is a macro that invokes the set position routine associated with the XDR stream xdrs. The parameter pos is a position value obtained from xdr_getpos(). This routine returns TRUE if the XDR stream could be repositioned, and FALSE otherwise. This routine is called indirectly via the pointer stored in the structure XDR. This routine may also be called with the upper-case XDR_SETPOS.

It is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

Parameters

xdrs	XDR stream.
pos	Position value obtained from xdr_getpos().

Files

rpc.h - RPC include file.

See Also

xdr_getpos().

xdr_short()

Translate between C short integers and their external representations.

Synopsis

```
bool_t xdr_short(xdrs, sp)
XDR      *xdrs;
short    *sp;
```

Description

xdr_short() is a filter primitive that translates between C short integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
sp	Pointer to short.

Files

rpc.h - RPC include file.

See Also

xdr_long().

xdr_string()

Translate between C strings and their external representations.

Synopsis

```
bool_t xdr_string(xdrs, sp, maxsize)
XDR        *xdrs;
char      **sp;
u_int       maxsize;
```

Description

xdr_string() is a filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than maxsize.

*sp is the address of the string's pointer. While decoding, if *sp is NULL, then the necessary storage is allocated to hold this null-terminated string and *sp is set to point to this. Use xdr_free() to free this storage.

This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
sp	Address of the string’s pointer.
maxsize	Maximum size of string.

Files

rpc.h - RPC include file

See Also

xdr_char().

xdr_u_char()

Translate between unsigned C chars and their external representations.

Synopsis

```
bool_t xdr_u_char(xdrs, ucp)
XDR             *xdrs;
unsigned char   *ucp;
```

Description

xdr_u_char() is a filter primitive that translates between unsigned C characters and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
ucp	XDR stream. Pointer to the unsigned character.

Files

rpc.h - RPC include file.

See Also

xdr_char().

xdr_u_int()

Translate between unsigned C integers and their external representations.

Synopsis

```
bool_t xdr_u_int(xdrs, up)
XDR            *xdrs;
unsigned int   *up;
```

Description

xdr_u_int() is a filter primitive that translates between C unsigned integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
up	Pointer to unsigned integer.

Files

rpc.h - RPC include file.

See Also

xdr_int()

xdr_u_long()

Translate between unsigned C long integers and their external representations.

Synopsis

```
bool_t xdr_u_long(xdrs, ulp)
XDR             *xdrs;
unsigned long   *ulp;
```

Description

xdr_u_long() is a filter primitive that translates between C unsigned long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
ulp	Pointer to the unsigned long integer.

Files

rpc.h - RPC include file.

See Also

xdr_int()

xdr_u_short()

Translate between unsigned C short integers and their external representations.

Synopsis

```
bool_t xdr_u_short(xdrs, usp)
XDR               *xdrs;
unsigned short    *usp;
```

Description

xdr_u_short() is a filter primitive that translates between C unsigned short integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
usp	Pointer to unsigned short integer.

Files

rpc.h - RPC include file.

See Also

xdr_int()

xdr_union()

Translate between discriminated C union and its external representation.

Synopsis

```
bool_t xdr_union(xdrs, dscmp, unp, choices, defaultarm)
XDR                  *xdrs;
int                  *dscmp;
char                 *unp
struct xdr_discrim   *choices;
bool_t               (*defaultarm)();    /* may equal NULL */
```

Description

xdr_union() is a filter primitive that translates between a discriminated C union and its external representation. It first translates the discriminant of the union located at dscmp. This discriminant is always an enum_t. Next the union located at unp is translated. The parameter choices is a pointer to an array of xdr_discrim() structures. Each structure contains an ordered pair of [value, proc]. If the union's discriminant is equal to the associated value, then the proc is called to translate the union. The end of the xdr_discrim() structure array is denoted by a routine value of NULL. If the discriminant is not found in the choices array, then the defaultarm procedure is called (if it is not NULL). This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
dscmp	Union discriminant.
unp	Address of union.
choices	Pointer to array of xdr_discrim() structures.
defaultarm	Procedure called if discriminant not found in choices array.

Files

rpc.h - RPC include file.

See Also

xdr_discrim()

xdr_vector()

Translate between fixed length arrays and their external representations.

Synopsis

```
bool_t xdr_vector(xdrs, arrp, size, elsize, elproc)
XDR          *xdrs;
char         *arrp;
u_int         size, elsize;
xdrproc_t     elproc;
```

Description

xdr_vector() is a filter primitive that translates between fixed length arrays and their external representations. The parameter arrp is the address of the pointer to the array, while size is the element count of the array. The parameter elsize is the size of each of the array's elements, and elproc is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
arrp	Address of the pointer to the array.
size	Element count of the array.
elsize	Size of each of the array’s elements.
elproc	XDR filter.

Files

rpc.h - RPC include file.

See Also

xdr_array()

xdr_void()

Routine that always returns one.

Synopsis

```
bool_t xdr_void()
```

Description

xdr_void() always returns one. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

Files

rpc.h - RPC include file

xdr_wrapstring()

Call xdr_string().

Synopsis

```
bool_t xdr_wrapstring(xdrs, sp)
XDR     *xdrs;
char    **sp;
```

Description

xdr_wrapstring() is a primitive that calls xdr_string(xdrs, sp, MAXUN.UNSIGNED); where MAXUN.UNSIGNED is the maximum value of an unsigned integer. xdr_wrapstring() is handy because the RPC package passes a maximum of two XDR routines as parameters and xdr_string(), one of the most frequently used primitives, requires three. Returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
sp	Address of the string.

Files

rpc.h - RPC include file.

See Also

xdr_array()

xdrmem_create()

Initialize XDR stream.

Synopsis

```
void xdrmem_create(xdrs, addr, size, op)
XDR             *xdrs;
char            *addr;
u_int            size;
enum xdr_op      op;
```

Description

xdrmem_create() initializes the XDR stream object pointed to by xdrs. The stream's data is written to, or read from, a chunk of memory at location addr whose length is no more than size bytes long. The op determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Parameters

xdrs	XDR stream.
addr	Object.
size	Length of object.
op	Direction of XDR stream.

Files

rpc.h - RPC include file.

See Also

xdr_bool(), xdr_bytes(), xdr_char(), xdr_opaque(), xdr_string().

xdrrec_create()

Initialize XDR stream object.

Synopsis

```
void xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
XDR       *xdrs;
u_int      sendsize, recvsize;
char      *handle;
int       (*readit)(), (*writeit)();
```

Description

xdrmem_create() initializes the XDR stream object pointed to by xdrs. The stream's data is written to a buffer of size sendsize; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size recvsize; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, writeit is called. Similarly, when a stream's input buffer is empty, readit is called. The behavior of these two routines is similar to the system calls read and write, except that handle is passed to the former routines as the first parameter.

Note: The XDR stream's op field must be set by the caller.

This XDR stream implements an intermediate record stream. Therefore, there are additional bytes in the stream to provide record boundary information.

Parameters

xdrs	XDR stream.
sendsize	Write buffer size.
recvsize	Read buffer size.
handle	Passed to redit and writeit routines.
readit	Called when input buffer is empty.
writeit	Called when output buffer is full.

Files

rpc.h - RPC include file.

See Also

read(), write()

xdrrec_endofrecord()

Mark data in buffer as completed record.

Synopsis

```
bool_t xdrrec_endofrecord(xdrs, sendnow)
XDR     *xdrs;
int      sendnow;
```

Description

xdrmem_endofrecord() can be invoked only on streams created by xdrrec_create(). The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if sendnow is non-zero. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.
sendnow	Write out data (if set).

Files

rpc.h - RPC include file.

See Also

xdrrec_create(), xdrrec_eof(), xdrrec_skiprecord()

xdrrec_eof()

Mark data in buffer as end of file.

Synopsis

```
bool_t xdrrec_eof(xdrs, empty)
XDR    *xdrs;
int     empty;
```

Description

xdrmem_eof() can be invoked only on streams created by xdrrec_create(). After consuming the rest of the current record in the stream, this routine returns TRUE if the stream has no more input, FALSE otherwise.

Parameters

xdrs	XDR stream.
empty	No more data.

Files

rpc.h - RPC include file.

See Also

xdrrec_create(), xdrrec_endofrecord(), xdrrec_skiprecord()

xdrrec_skiprecord()

Discard rest of current record.

Synopsis

```
bool_t xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

Description

xdrmem_skiprecord() can be invoked only on streams created by xdrrec_create(). It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns TRUE if it succeeds, FALSE otherwise.

Parameters

xdrs	XDR stream.

Files

rpc.h - RPC include file.

See Also

xdrrec_create(), xdrrec_endofrecord(), xdrrec_eof().

xdrstdio_create()

Initialize XDR stream.

Synopsis

```
void xdrstdio_create(xdrs, file, op)
XDR          *xdrs;
FILE         *file;
enum xdr_op   op;
```

Description

xdrstdio_create() initializes the XDR stream object pointed to by xdrs. The XDR stream data is written to, or read from, the standard I/O stream file. The parameter op determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Note: The destroy routine associated with such XDR streams calls fflush() on the file stream, but never fclose().

Parameters

xdrs	XDR stream.
file	Standard I/O stream.
op	Direction of XDR stream.

Files

rpc.h - RPC include file.

See Also

xdrmem_create(), xdrrec_create().

Appendix

C

RPC Library Header Files

This appendix describes the RPC library header files. It lists the header files used by the RPC library.

Header Files

You should include rpc.h, which pulls in all the necessary header files for an RPC application program. If the user wants to interface to the portmapper, pmapclnt.h must be included explicitly.

The header files used by the RPC library are listed in the following table:

Header File	Description
auth.h	Interface to generic authentication routines included by rpc.h automatically.
authunix.h	Interface to UNIX type authentication routines included by rpc.h automatically.
clnt.h	Interface to routines required by RPC clients included by rpc.h automatically.
pmapclnt.h	Interface to portmapper services for clients and servers.
pmapprot.h	Describes the portmapper protocol.
pmaprmt.h	Interface to portmapper remote call service.

Header File	Description
rpc.h	Includes the necessary header files for an RPC client or server. This is the main header file that must be included by user programs. It pulls in these header files: ■ xdrtypes.h ■ auth.h ■ authunix.h ■ svcauth.h ■ inet.h ■ clnt.h ■ rpcmsc.h ■ netdb.h ■ xdr.h ■ rpcmsg.h ■ svc.h
rpcget.h	Internal interface file required by routines of the RPC library. / This file is not required by users to interface to the RPC library.
rpcmsc.h	Internal interface header file for miscellaneous routines; included by rpc.h automatically.
rpcmsg.h	Definition of the RPC message format included by rpc.h automatically.
svc.h	Interface required by RPC servers; included by rpc.h automatically (Also see svcrpc.h). Due to a conflicting include filename for the SAS/C compiler, the file is renamed to svcrpc.h.
svcauth.h	Interface to server side RPC authentication included by rpc.h automatically.
svccall.h	Internal interface definitions not seen externally by the RPC library.
svcrpc.h	Interface required by RPC servers; included by rpc.h automatically. This file is normally called svc.h. Due to a conflicting include filename for the SAS/C compiler, this include file has been renamed to svcrpc.h.

Header File	Description
svctcp.h	Internal interface definition for RPC servers using TCP as the transport mechanism. Not required by users of the RPC library.
xrd.h	Interface definition of external data representation serialization routines; included by rpc.h automatically.
xdrrst.h	Internal interface definition for record formatting when using a byte stream protocol such as TCP. Not required to be included by a user's RPC program.
xdrtypes.h	Addition types required by the RPC library routines and definitions of RPC error message numbers; included by rpc.h automatically.

Appendix

D RPC Log

This appendix provides reference information on RPC Log. It describes the externally defined function rpclog().

RPC Log Interface

When an error is detected by the RPC library, it calls an externally defined function called rpclog(). The default rpclog() shipped with the RPC library simply formats the information passed to it and then prints it to stderr.

Source for Default rpclog

The following is the source for the default rpclog():

```
#include <stdio.h>
#include <rpc.h>

void rpclog(number, csectp, funcp, msgp)
int     number;
char   *csectp;
char   *funcp;
char   *msgp;
{
    fprintf(stderr, "%s%s\n", funcp, msgp);
    return;
}
```

number	The error number (defined in xdrtype.h).
csectp	A pointer to a string defining the csect that encountered the error and called rpclog().
funcp	A string defining the function that encountered the error and called rpclog().
msgp	A pointer to the message text.

If the default rpclog() function does not suffice for your application, it may be replaced by an application specific function. The new function should allow for the same calling sequence and return a void.

Appendix

E

Sample JCL

This appendix includes sample JCL. It includes these sections:

- Nonreentrant User Program: C/370 Compiler—Compile and link a nonreentrant user program using the C socket library, the RPC library and the C/370 compiler
- Reentrant User Program: C/370 Compiler—Compile and link a reentrant user program using the C socket library, the RPC library and the C/370 compiler
- Nonreentrant User Program: SAS/C Compiler—Compile and link a nonreentrant user program using the C socket library, the RPC library and the SAS/C compiler
- Reentrant User Program: SAS/C Compiler—Compile and link a reentrant user program using the C socket library, the RPC library and the SAS/C compiler

Note: If you are link-editing with the BINDER (HEWLF096) under SMP/E, you may get the error message IEW2480W. This message can be safely ignored. You can turn this message off by setting option MSGLEVEL=4 in the PARM field of the linkedit (binder).

Nonreentrant User Program: C/370 Compiler

```
//RPCIBMC  JOB
//*
//*   SAMPLE JCL TO COMPILE, LINK, AND EXECUTE A NONREENTRANT
//*   USER PROGRAM USING THE TCP/API C SOCKET LIBRARY, THE
//*   TCP/API RPC/XDR LIBRARY, AND THE IBM C/370 C COMPILER.
//*
//*   EDIT THE JOB JCL STATEMENT, VERIFY THE DATA SET NAME(S)
//*   OF THE USER'S DATA SETS, AND VERIFY THAT THE DATA SET
//*   NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
//*   FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED
//*   ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB ASSUMES
//*   THAT THE STANDARD IBM C/370 EDCCLG JCL PROCEDURE IS
//*   AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
//*
//CLGNRENT EXEC EDCCLG,
//      INFILE='USER.C(CPROG)             <=== VERIFY DSNAME
//      PARM='NORENT,DEF(IBMC)',
//      GPARM='PROGRAM PARAMETERS'        <=== VERIFY PARAMETERS
//*
//*   INCLUDE THE TCP/API SOCKET INCLUDE (.H) DATA SET IN THE
//*   COMPILER SYSLIB CONCATENATION. BOTH THE SOCKET AND RPC/XDR
//*   INCLUDE FILES ARE FOUND IN THE SAME SYSLIB DATA SET.
//*
//COMPILE.SYSLIB    DD DISP=SHR,DSN=TRGINDX.H                  <=== VERIFY DSNAME
//                  DD DISP=SHR,DSN=&VSCCHD&CVER&EDCHDRS
//*   INCLUDE THE TCP/API SOCKET SUBROUTINE LIBRARY DATA SET IN
//*   THE LINKAGE EDITOR SYSLIB CONCATENATION. BOTH THE SOCKET
//*   AND RPC/XDR LOAD MODULES ARE INCLUDED IN THE SAME SYSLIB
//*   DATA SET.
//*
//LKED.SYSLIB       DD
//                  DD
//                  DD DISP=SHR,DSN=TRGINDX.CILIB               <=== VERIFY
DSNAME
//
//LKED.SYSIN        DD DUMMY,DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120)
```

Reentrant User Program: C/370 Compiler

This JCL is for the C/370 version of the compiler.

If you are using the Ad/Cycle compiler, replace the following line (shown in bold in the JCL):

```
//   DD DISP=SHR,DSN=&VSCCHD&CVER&EDCHDRS
```

with this line:

```
//   DD DISP=SHR,DSN=&LNGPRFX..SEDCDHDR
```

```
//RPCIBMCR JOB
//*
//*   SAMPLE JCL TO COMPILE, LINK, AND EXECUTE A REENTRANT USER
//*   PROGRAM USING THE TCP/API C SOCKET LIBRARY, THE TCP/API
//*   RPC/XDR LIBRARY, AND THE IBM C/370 C COMPILER.
//*
//*   EDIT THE JOB JCL STATEMENT, VERIFY THE DATA SET NAME(S)
//*   OF THE USER'S DATA SETS, AND VERIFY THAT THE DATA SET
//*   NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
//*   FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED ARE
//*   MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB ASSUMES THAT
//*   THE STANDARD IBM C/370 EDCCPLG JCL PROCEDURE IS AVAILABLE
//*   IN AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
//*
//CLGRENT  EXEC EDCCPLG,
//         INFILE='USER.C(CPROG)',            <=== VERIFY DSNAME
//         CPARM='RENT,DEF(IBMC)',
//         GPARM='PROGRAM PARAMETERS'      <=== VERIFY PARAMETERS
//*
//*   INCLUDE THE TCP/API SOCKET INCLUDE (.H) DATA SET IN THE
//*   COMPILER SYSLIB CONCATENATION. BOTH THE SOCKET AND RPC/XDR
//*   INCLUDE FILES ARE FOUND IN THE SAME SYSLIB DATA SET.
//*
//COMPILE.SYSLIB   DD DISP=SHR,DSN=TRGINDX.H   <=== VERIFY DSNAME
//         DD DISP=SHR,DSN=&VSCCHD&CVER&EDCHDRS <=== Ad/Cycle name
//*
//*   INCLUDE THE TCP/API SOCKET SUBROUTINE OBJECT LIBRARY
//*   DATA SET IN THE PREPROCESSOR SYSLIB CONCATENATION.
//*   BOTH THE SOCKET AND RPC/XDR OBJECT MODULES ARE INCLUDED
//*   IN THE SAME SYSLIB DATA SET.
//*
//PLKED.SYSLIB       DD DISP=SHR,
//                   DSN=TRGINDX.CIROBJ        <=== VERIFY DSNAME
//PLKED.SYSIN        DD DSN=*.COMPILE.SYSLIN,DISP=(OLD,DELETE)
//                   DD *
 INCLUDE SYSLIB(S0SKCF)
 INCLUDE SYSLIB(S0INTR)
 INCLUDE SYSLIB(RPCFDS)
 ENTRY   CEESTART
/*
//
```

Nonreentrant User Program: SAS/C Compiler

```
//RPCSASC  JOB
//*
//*   SAMPLE JCL TO COMPILE, LINK, AND EXECUTE A NONREENTRANT
//*   USER PROGRAM USING THE TCP/API C SOCKET LIBRARY, THE
//*   TCP/API C RPC/XDR LIBRARY, AND THE SAS/C COMPILER. THIS
//*   SAMPLE WORKS WITH SAS/C 4.50, 5.00, AND 5.01.
//*
//*   EDIT THE JOB JCL STATEMENT, VERIFY THE DATA SET NAME(S)
//*   OF THE USER'S DATA SETS, AND VERIFY THAT THE DATA SET
//*   NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
//*   FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED
//*   ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB ASSUMES
//*   THAT THE STANDARD SAS SAS/C LC370CLG JCL PROCEDURE IS
//*   AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
//*
//CLNORENT EXEC LC370CLG,
//   PARM.C='NORENT,DEF(SASC)',
//   PARM.GO='PROGRAM PARAMETERS'                                  <=== VERIFY
PARAMETERS
//*
//*   TCP/IP INCLUDE FILE DATA SET MUST PRECEDE SAS/C
//*   DATA SET. BOTH THE SOCKET AND RPC/XDR INCLUDE FILES
//*   ARE IN THE SAME DATA SET.
//*
//C.SYSLIB    DD  DISP=SHR,DSN=TRGINDX.H       <=== VERIFY DSNAME
//            DD  DISP=SHR,DSN=&MACLIB
//C.SYSIN     DD  DISP=SHR,
//                DSN=USER.C(CPROG)         <=== VERIFY DSNAME
//*
//*   TCP/IP OBJECT SYSLIB DATA SET MUST PRECEDE SAS/C
//*   DATA SETS. BOTH THE SOCKET AND RPC/XDR LOAD
//*   MODULES ARE IN THE SAME SYSLIB DATA SET.
//*
//LKED.SYSLIB  DD  DISP=SHR,
//                 DSN=TRGINDX.CSLIB         <=== VERIFY DSNAME
//             DD  DISP=SHR,DSN=SASC.&ENV.LIB  <=== VERIFY DSNAME
//             DD  DISP=SHR,DSN=&SYSLIB
//             DD  DISP=SHR,DSN=&CALLLIB
//LKED.SYSIN   DD  DUMMY,DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120)
```

Reentrant User Program: SAS/C Compiler

```
//RPCSASCR JOB
//*
//*   SAMPLE JCL TO COMPILE, LINK, AND EXECUTE A REENTRANT C
//*   PROGRAM USING THE TCP/API C SOCKET LIBRARY, THE TCP/IP
//*   C RPC/XDR LIBRARY, AND THE SAS/C C COMPILER. THIS SAMPLE
//*   WORKS WITH SAS/C 4.50, 5.00, AND 5.01.
//*
//*   EDIT THE JOB JCL STATEMENT, VERIFY THE DATA SET NAME(S)
//*   OF THE USER'S DATA SETS, AND VERIFY THAT THE DATA SET
//*   NAMES REFERENCED BELOW MATCH THE NAMES THAT YOU SELECTED
//*   FOR THE TCP/IP TARGET DATA SETS (DSN'S TO BE VERIFIED
//*   ARE MARKED BELOW WITH "<=== VERIFY ..."). THIS JOB ASSUMES
//*   THAT THE STANDARD SAS SAS/C LC370C AND LC370LRG JCL
//*   PROCEDURES ARE AVAILABLE IN YOUR INSTALLATION'S PROCLIB(S).
//*   STEP 1: COMPILE USER PROGRAM REENTRANTLY.
//*
//CCRENT   EXEC LC370C,
//   PARM.C='RENT,DEF(SASC)'
//*
//*   TCP/IP INCLUDE FILE DATA SET MUST PRECEDE SAS/C
//*   DATA SET. BOTH THE SOCKET AND RPC/XDR INCLUDE FILES
//*   ARE IN THE SAME DATA SET.
//*
//C.SYSLIB    DD  DISP=SHR,DSN=TRGINDX.H    <=== VERIFY DSNAME
//            DD  DISP=SHR,DSN=&MACLIB
//C.SYSIN     DD  DISP=SHR,
//            DSN=USER.C(CPROG)             <=== VERIFY DSNAME
//*
//*   STEP 2: LINK USER PROGRAM USING SAS/C CLINK
//*   PREPROCESSOR AND THEN EXECUTE.
//*
//LKRENT   EXEC LC370LRG,PARM.LKED='LIST,MAP,RENT',
//   PARM.GO='PROGRAM PARAMETERS'          <=== VERIFY PARAMETERS
//*
//*   TCP/IP OBJECT SYSLIB DATA SET MUST PRECEDE SAS/C
//*   DATA SETS. BOTH THE SOCKET AND RPC/XDR LOAD MODULES
//*   ARE IN THE SAME SYSLIB DATA SET.
//*
//*   TCP/IP OBJECT SYSLIB DATA SET MUST PRECEDE SAS/C
//*   DATA SETS. BOTH THE SOCKET AND RPC/XDR LOAD MODULES
//*   ARE IN THE SAME SYSLIB DATA SET.
//*
//LKED.SYSLIB  DD  DISP=SHR,DSN=TRGINDX.CSROBJ <=== VERIFY DSNAME
//             DD  DDNAME=AR#&ALLRES
//             DD  DISP=SHR,DSN=SASC.&ENV.OBJ  <=== VERIFY DSNAME
//             DD  DISP=SHR,DSN=&SYSLIB
//             DD  DISP=SHR,DSN=&CALLLIB
//LKED.SYSIN   DD  DISP=(OLD,DELETE),DSN=*.CCRENT.C.SYSLIN
//             DD *
 INCLUDE SYSLIB(S0SKCF)
 INCLUDE SYSLIB(S0INTR)
 INCLUDE SYSLIB(RPCFDS)
 ENTRY MAIN
//
```

Appendix

F

Sample RPC Programs

This appendix contains source code listings for the four sample programs provided as part of the RPC library. It includes the following sections:

- Sample Programs—Provides an overview of the sample programs, including instructions on execution
- Sample Programs' Source Code—Includes the code for the msgsvc, msgclnt, sortsvc, and sortclnt sample programs

Four sample programs are provided as part of the RPC library. They are paired into two sets with each set having a client and server program. The client of the first pair, using the RPC library, simply sends a user selected message to the server, which then prints the message to stderr.

The client of the second pair sends a user selected group of arguments to the server via RPC. The server then sorts the arguments and returns the results to the client, which then prints the sorted result to stderr.

The programs can be built using the sample JCL provided to build any RPC library program. Once built, the server program should be started and then the corresponding client should be started.

Sample Programs

The following sections tell you how to run the sample message and sort programs.

To Run the Sample Message Programs

Execute the server msgsvc by calling it from TSO or batch with no parameters

```
CALL 'MYLOAD(MSGSVC)'
```

Then execute the client, msgcln, by calling it from TSO or batch with the name of the host on which the server is running and then a message. (Use the appropriate C compiler parameter passing conventions.)

```
CALL 'MYLOAD(MSGCLNT)' 'SERVERHOST HELLO'
```

To Run the Sample Sort Programs

Execute the server, sortsvc, by calling it from TSO or batch with no parameters.

```
CALL 'MYLOAD(SORTSVC)'
```

Then execute the client, sortclnt by calling it from TSO or batch with the name of the host on which the server is running and then any number of string arguments to be sorted. (Use the appropriate C compiler parameter passing conventions.)

```
CALL 'MYLOAD(SORTCLNT)' 'SERVERHOST D E C B A'
```

Sample Programs' Source Code

This section includes the source code listings for the sample programs.

MSGSVC

The following is the source code listing for the msgsvc program:

```
#include <stdio.h>
#include <rpc.h>
#include <pmapclnt.h>
#include <msg.h>

static void   messageprog_1();
static char   *printmessage_1();

static struct timeval TIMEOUT = { 25, 0 };

main()
{
    SVCXPRT *transp;

    (void)pmap_unset(MESSAGEPROG, MESSAGEVERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == (SVCXPRT *)NULL)
    {
        (void)fprintf(stderr, "CANNOT CREATE UDP SERVICE.\n");
        exit(16);
    }
    if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS,
           messageprog_1, IPPROTO_UDP))
    {
        (void)fprintf(stderr,
        "UNABLE TO REGISTER (MESSAGEPROG, MESSAGEVERS, UDP).\n");
        exit(16);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == (SVCXPRT *)NULL)
    {
        (void)fprintf(stderr, "CANNOT CREATE TCP SERVICE.\n");
        exit(16);
    }
    if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS,
                  messageprog_1, IPPROTO_TCP))
    {
        (void)fprintf(stderr,
        "UNABLE TO REGISTER (MESSAGEPROG, MESSAGEVERS, TCP).\n");
        exit(16);
    }
    svc_run();
    (void)fprintf(stderr, "SVC_RUN RETURNED\n");
    exit(16);
    return(0);
}

static void messageprog_1(rqstp, transp)
struct svc_req   *rqstp;
SVCXPRT  *transp;
```

```
{
    union
    {
        char  *printmessage_1_arg;
    }
    argument;
    char      *result;
    bool_t  (*xdr_argument)();
    bool_t  (*xdr_result)();
    char   *(*local)();

    switch (rqstp->rq_proc)
    {
        case NULLPROC:
            (void)svc_sendreply(transp, xdr_void, (char *)NULL);
            return;

        case PRINTMESSAGE:
            xdr_argument = xdr_wrapstring;
            xdr_result = xdr_int;
            local = (char *(*)()) printmessage_1;
            break;

        default:
            svcerr_noproc(transp);
            return;
    }
    bzero((char *)&argument, sizeof(argument));
    if (!svc_getargs(transp, xdr_argument, &argument))
    {
        svcerr_decode(transp);
        return;
    }
    result = (*local)(&argument, rqstp);
    if (result != (char *)NULL &&
        !svc_sendreply(transp, xdr_result, result))
    {
        svcerr_systemerr(transp);
    }
    if (!svc_freeargs(transp, xdr_argument, &argument))
    {
        (void)fprintf(stderr, "UNABLE TO FREE ARGUMENTS\n");
        exit(16);
    }
    return;
}

char  *printmessage_1(msg)
char  **msg;
{
    static char  result;

    fprintf(stderr, "%s\n", *msg);
    result = 1;
    return(&result);
}
```

MSGCLNT

The following is the source listing for the msgclnt program:

```
/* @(#)rprintmsg.c 2.1 88/08/11 4.0 RPCSRC */
#include <stdio.h>
#include <rpc.h>
#include <time.h>
#include <msg.h>

static struct timeval TIMEOUT = { 25, 0 };
 static int   *printmessage_1();
 main(argc, argv)
int   argc;
char  *argv[];
{
    CLIENT   *cl;
    int      *result;
    char     *server ;
    char     *message;
    if (argc < 3)
    {
        fprintf(stderr, "USAGE: %s HOST MESSAGE\n", argv[0]);
        exit(16);
    }
    server  = argv[1];
    message = argv[2];

    cl = clnt_create(server , MESSAGEPROG, MESSAGEVERS, "tcp");
    if (cl == (CLIENT *)NULL)
    {
        clnt_pcreateerror(server );
        exit(16);
    }

    result = printmessage_1(&message, cl);
    if (result == (int *)NULL)
    {
        clnt_perror(cl, server );
        exit(16);
    }

    if (*result == 0)
    {
        fprintf(stderr, "%s: SORRY, %s COULDN'T PRINT YOUR MESSAGE\n",
            argv[0], server );
        exit(16);
    }
    printf("MESSAGE DELIVERED TO %s!\n", server );
    exit(0);
    return(0);
```

```
}

static int *printmessage_1(argp, clnt)
char    **argp;
CLIENT  *clnt;
{
    static int res;

    bzero((char *)&res, sizeof(res));
    if (clnt_call(clnt, PRINTMESSAGE, xdr_wrapstring,
        argp, xdr_int, &res, TIMEOUT) != RPC_SUCCESS)
    {
        return ((int *)NULL);
    }
    return (&res);
}
SORTSVC
This is the source code listing for the sortsvc program:
#include <stdio.h>
#include <rpc.h>
#include <pmapclnt.h>
#include <sort.h>

static void                 sortprog_1();
static bool_t               xdr_str();
static bool_t               xdr_sortstrings();
static int                  comparestrings();
static struct sortstrings   *sort_1();

main()
{
    SVCXPRT    *transp;

    (void)pmap_unset(SORTPROG, SORTVERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == (SVCXPRT *)NULL)
    {
        (void)fprintf(stderr, "CANNOT CREATE UDP SERVICE.\n");
        exit(16);
    }
    if (!svc_register(transp, SORTPROG, SORTVERS, sortprog_1,
```

```
                IPPROTO_UDP))
    {
        (void)fprintf(stderr,
        "UNABLE TO REGISTER (SORTPROG, SORTVERS, UDP).\n");
        exit(16);
    }
    svc_run();
    (void)fprintf(stderr, "SVC_RUN RETURNED\n");
    exit(16);
    return(0);
}

static void sortprog_1(rqstp, transp)
struct svc_req   *rqstp;
SVCXPRT          *transp;
{
    union
    {
        sortstrings sort_1_arg;
    }
    argument;

    char      *result;
    bool_t    (*xdr_argument)();
    bool_t    (*xdr_result)();
    char      *(*local)();


    switch (rqstp->rq_proc)
    {
    case NULLPROC:
        (void)svc_sendreply(transp, xdr_void, (char *)NULL);
        return;

    case SORT:
        xdr_argument = xdr_sortstrings;
        xdr_result = xdr_sortstrings;
        local = (char *(*)()) sort_1;
        break;

    default:
        svcerr_noproc(transp);
        return;
    }
    bzero((char *)&argument, sizeof(argument));
    if (!svc_getargs(transp, xdr_argument, &argument))
    {
        svcerr_decode(transp);
        return;
    }
    result = (*local)(&argument, rqstp);
    if (result != (char *)NULL &&
        !svc_sendreply(transp, xdr_result, result))
    {
        svcerr_systemerr(transp);
    }
    if (!svc_freeargs(transp, xdr_argument, &argument))
    {
        (void)fprintf(stderr, "UNABLE TO FREE ARGUMENTS\n");
        exit(16);
    }
    return;
}
```

```
static int comparestrings(sp1, sp2)
char   **sp1;
char   **sp2;
{
    return (strcmp(*sp1, *sp2));
}

static struct sortstrings  *sort_1(ssp)
struct sortstrings         *ssp;
{
    static struct sortstrings  ss_res;

    if (ss_res.ss.ss_val != (str *)NULL)
    {
        free(ss_res.ss.ss_val);
    }

    qsort(ssp->ss.ss_val, ssp->ss.ss_len, sizeof(char *), comparestrings);
    ss_res.ss.ss_len = ssp->ss.ss_len;
    ss_res.ss.ss_val =
         (str *)malloc(ssp->ss.ss_len * sizeof(str *));
    bcopy(ssp->ss.ss_val, ss_res.ss.ss_val,
          ssp->ss.ss_len * sizeof(str *));
    return(&ss_res);
}

static bool_t xdr_str(xdrs, objp)
XDR        *xdrs;
str        *objp;


{
    if (!xdr_string(xdrs, objp, MAXSTRINGLEN))
    {
        return (FALSE);
    }
    return (TRUE);
}

static bool_t xdr_sortstrings(xdrs, objp)
XDR            *xdrs;
sortstrings    *objp;
{
    if (!xdr_array(
                   xdrs, (char **)&objp->ss.ss_val,
                   (u_int *)&objp->ss.ss_len, MAXSORTSIZE,
                   sizeof(str), xdr_str))
    {
        return (FALSE);
    }
    return (TRUE);
}
```

SORTCLNT

This is the source code listing for the sortclnt program:

```
/* @(#)rsort.c 2.1 88/08/11 4.0 RPCSRC */
#include <stdio.h>
#include <rpc.h>
#include <sort.h>

static bool_t        xdr_sortstrings();
static bool_t        xdr_str();
static sortstrings   *sort_1();
main(argc, argv)
int     argc;
char    **argv;
{
    char                 *machinename;
    struct sortstrings   args;
    struct sortstrings   res;
    int                  i;
    if (argc < 3)
    {
        fprintf(stderr, "USAGE: %s MACHINENAME {S1 ...}\n", argv[0]);
        exit(16);
    }
    machinename = argv[1];
    args.ss.ss_len = argc - 2;
    args.ss.ss_val = &argv[2];
    res.ss.ss_val = (char **)NULL;
    if ((i = callrpc(machinename, SORTPROG, SORTVERS, SORT,
   xdr_sortstrings, (char *)&args, xdr_sortstrings, (char *)&res)))
    {
        fprintf(stderr, "%s: CALL TO SORT SERVICE FAILED. ", argv[0]);
        clnt_perrno(i);
        fprintf(stderr, "\n");
        exit(16);
```

```
    }

    for (i = 0; i < res.ss.ss_len; i++)
    {
        printf("%s\n", res.ss.ss_val[i]);
    }

    exit(0);
    return(0);
}


static struct timeval TIMEOUT = { 25, 0 };

static sortstrings   *sort_1(argp, clnt)
sortstrings          *argp;
CLIENT               *clnt;
{
    static sortstrings    res;

    bzero((char *)&res, sizeof(res));
    if (clnt_call(clnt, SORT, xdr_sortstrings, argp,
        xdr_sortstrings, &res, TIMEOUT) != RPC_SUCCESS)
    {
        return ((sortstrings *)NULL);
    }
    return (&res);
}


static bool_t xdr_str(xdrs, objp)
XDR    *xdrs;
str    *objp;
{
    if (!xdr_string(xdrs, objp, MAXSTRINGLEN))
    {
        return (FALSE);
    }
    return (TRUE);
}

static bool_t xdr_sortstrings(xdrs, objp)
XDR           *xdrs;
sortstrings   *objp;
{
    if (!xdr_array(
                  xdrs, (char **)&objp->ss.ss_val,
                  (u_int *)&objp->ss.ss_len, MAXSORTSIZE,
                  sizeof(str), xdr_str))
    {
        return (FALSE);
    }
    return (TRUE);
}
```

Index

#

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

X

Computer Associates™